TURÎNG

图灵程序设计丛书

CPU航

【日】水头一寿 米泽辽 藤田裕士 著 赵谦 译



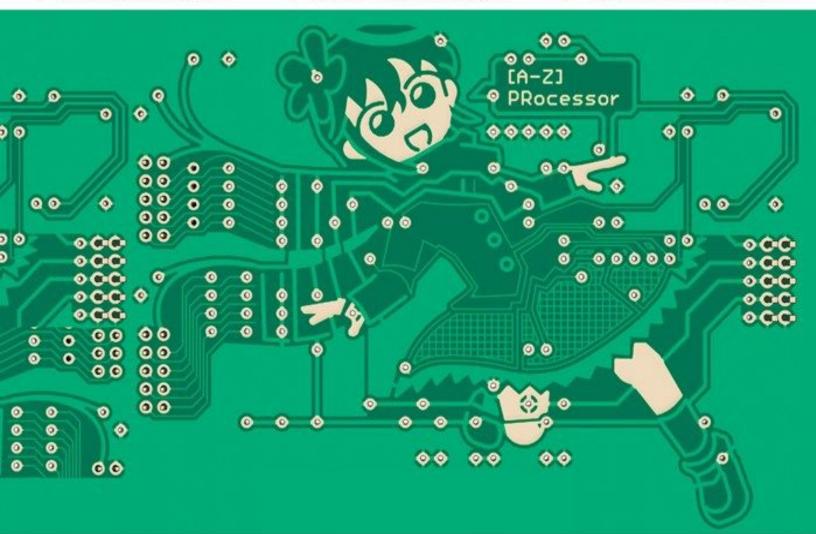
只需编程基础 从零开始设计和实现CPU



CPU、总线、内存、I/O 组合成一个简单的计算机系统



超强实践性
从硬件到软件,统统自己动手





版权信息

书名: CPU自制入门

作者: (日)水头一寿, (日)米泽辽, (日)藤田裕士

译者: 赵谦

本书由北京图灵文化发展有限公司发行数字版。版权所有,侵权必究。

您购买的图灵电子书仅供您个人使用,未经授权,不得以任何方式复制 和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟,与我们共同保护知识产权。

如果购买者有侵权行为,我们可能对该用户实施包括但不限于关闭该帐号等维权措施,并可能追究法律责任。

图灵社区会员 ptpress (libowen@ptpress.com.cn) 专享 尊重版权

版权声明

译者序

声明

作者序

本书的阅读方法

第1章 CPU的设计与实现

- 1.1 序
- 1.2 计算机系统
 - 1.2.1 什么是计算机
 - 1.2.2 什么是 CPU
 - 1.2.3 什么是内存
 - 1.2.4 什么是 I/O
 - 1.2.5 什么是总线
 - 1.2.6 小结
- 1.3 数字电路基础
 - 1.3.1 什么是数字电路
 - 1.3.2 数值表达
 - 1.3.3 有符号二进制数
 - 1.3.4 MOSFET 的结构
 - 1.3.5 逻辑运算
 - 1.3.6 CMOS 基本逻辑门电路
 - 1.3.7 存储元件
 - 1.3.8 组合电路和时序电路
 - 1.3.9 时钟同步设计
 - 1.3.10 小结
- 1.4 Verilog HDL 语言
 - 1.4.1 什么是 Verilog HDL
 - 1.4.2 电路描述

- 1.4.3 电路仿真
- 1.4.4 Verilog HDL 的仿真环境
- 1.4.5 小结
- 1.5 系统蓝图
 - 1.5.1 目标系统整体介绍
 - 1.5.2 关于本章中的代码
- 1.6 总线的设计与实现
 - 1.6.1 总线的设计
 - 1.6.2 总线的实现
 - 1.6.3 小结
- 1.7 存储器的设计与实现
 - 1.7.1 FPGA 的 RAM 区域
 - 1.7.2 ROM 的设计与实现
 - 1.7.3 小结
- 1.8 AZ Processor 的设计与实现
 - 1.8.1 关于 CPU
 - 1.8.2 AZ Processor 的设计
 - 1.8.3 AZ Processor 的实现
 - 1.8.4 小结
- 1.9 I/O 的设计与实现
 - 1.9.1 定时器
 - 1.9.2 UART
 - 1.9.3 GPIO
 - 1.9.4 小结
- 1.10 AZPR SoC 整体连接
 - 1.10.1 各模块的连接
 - 1.10.2 时钟模块的实现
 - 1.10.3 顶层模块的实现

- 1.10.4 小结
- 1.11 AZPR SoC 的仿真
 - 1.11.1 仿真模型的编写
 - 1.11.2 Testbench 的编写
 - 1.11.3 执行仿真
 - 1.11.4 小结
- 1.12 本章总结
- 第2章 电路板的设计与制作
 - 2.1 序
 - 2.2 电路板规格
 - 2.2.1 电路板名称
 - 2.2.2 电路板的构成
 - 2.2.3 电路板尺寸
 - 2.2.4 电路板层数
 - 2.2.5 FPGA 选型
 - 2.2.6 外围电路的选定
 - 2.3 元件选型
 - 2.3.1 元件选型标准
 - 2.3.2 元件选型
 - 2.3.3 元件的选购
 - 2.4 电路设计
 - 2.4.1 下载规格书
 - 2.4.2 配置电路
 - 2.4.3 外围电路
 - 2.4.4 电源电路
 - 2.4.5 电路板设计环境
 - 2.4.6 使用 Eagle 设计电路图
 - 2.4.7 完成的电路图

- 2.5 布局设计
 - 2.5.1 电路板设计约束条件及布线策略
 - 2.5.2 FPGA 板的布局设计
 - 2.5.3 电源板的布局设计
 - 2.5.4 使用 Eagle 布局
 - 2.5.5 完成的布局
- 2.6 制作元件库
 - 2.6.1 制作 Symbol
 - 2.6.2 制作 Package
 - 2.6.3 制作 Device
- 2.7 电路板 3D 模型
 - 2.7.1 软件使用说明
 - 2.7.2 准备 3D 模型库
 - 2.7.3 制作电路板模型
- 2.8 制作感光电路板
 - 2.8.1 整体流程
 - 2.8.2 制作光罩
 - 2.8.3 粘合光罩
 - 2.8.4 曝光
 - 2.8.5 显像
 - 2.8.6 蚀刻
 - 2.8.7 阻焊剂
 - 2.8.8 开孔
 - 2.8.9 在背面安装 VPort 接头时的处理
 - 2.8.10 制作通孔
 - 2.8.11 飞线
- 2.9 使用电路板制造服务
 - 2.9.1 电路板制造服务

- 2.9.2 DRC
- 2.9.3 输出 Gerber 数据
- 2.9.4 检查 Gerber 数据
- 2.9.5 向 P 板 .com 公司下单制板
- 2.9.6 向 OLIMEX 公司下单制板
- 2.10 组装电路板
 - 2.10.1 电源板
 - 2.10.2 组装 FPGA 板
- 2.11 功能测试
 - 2.11.1 识别 FPGA
 - 2.11.2 诊断程序
- 2.12 本章总结

第3章 编程

- 3.1 序
- 3.2 开发环境
 - 3.2.1 准备工作
 - 3.2.2 FPGA 开发环境
 - 3.2.3 ISE WebPACK
 - 3.2.4 UrJTAG
 - 3.2.5 交叉汇编程序
 - 3.2.6 第一个程序
- 3.3 串口通信
 - 3.3.1 安装 Tera Term
 - 3.3.2 编写程序
 - 3.3.3 执行程序
- 3.4 程序加载器
 - 3.4.1 XMODEM 协议
 - 3.4.2 编写程序

- 3.4.3 编写加载测试程序
- 3.4.4 执行程序
- 3.5 中断与异常
 - 3.5.1 什么是中断
 - 3.5.2 编写程序
 - 3.5.3 执行程序
 - 3.5.4 什么是异常
 - 3.5.5 编写程序
 - 3.5.6 执行程序
- 3.6 七段数码管
 - 3.6.1 什么是七段数码管
 - 3.6.2 七段数码管的控制
 - 3.6.3 七段数码管计数器概要
 - 3.6.4 编写程序
 - 3.6.5 执行程序
- 3.7 制作一个实用程序
 - 3.7.1 功能概要
 - 3.7.2 制作程序
 - 3.7.3 执行程序
- 3.8 结语

版权声明

CPU JISAKU NYUMON by Kazutoshi Suito, Ryo Yonezawa, Yuji Fujita Copyright©2012 Kazutoshi Suito, Ryo Yonezawa, Yuji Fujita All rights reserved.

Original Japanese edition published by Gijyutsu-Hyoron Co., Ltd., Tokyo

This Simplified Chinese language edition published by arrangement with Gijyutsu-Hyoron Co., Ltd., Tokyo in care of Tuttle-Mori Agency, Inc., Tokyo

本书中文简体字版由 Gijyutsu-Hyoron Co., Ltd. 授权人民邮电出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

(图灵公司感谢李典对本书的审读。)

译者序

接触 IT 行业十多年来,我的书架上始终缺少一本书。我有各种语言的经典书籍和实用手册,它们帮助我使用最合适的工具解决问题。我还有一些操作系统、编译器和软件架构方面的书籍,它们指导我写出更高效的代码。然而对于操作系统之下的 CPU 内部世界,我的认识依然停留在大学时 80×86 处理器的课堂上。那门课让我学会了如何使用 CPU,而如何设计和实现 CPU 却始终是我知识体系中缺失的最底层的一环。

《CPU 自制入门》正是我一直寻找的那本书。本书介绍了计算机系统最物理、最底层的 3 个部分: CPU 设计制作、电路板设计制造以及汇编编程。作者们利用 FPGA 芯片,开启了一个崭新的自制 CPU 的世界。将如此广泛的技术内容以实践的方式结成一册,该书可谓首屈一指。

更让我印象深刻的是本书的阅读门槛非常低。几乎所有必要的基础知识书中都有介绍,如数字电路设计、Verilog语言,甚至还包括电路板CAD软件的使用,等等。其中任何一个内容展开讨论都需要几本书的篇幅,然而本书作者们却可以依靠丰富的经验,以最精简的文字,将最核心的知识汇集到一本书中,使各种知识背景的读者都可以方便地阅读。

近年来,随着摩尔定律接近极限,计算机系统很难再像从前那样单纯依靠芯片制程的进步获取速度提升。而为了设计更加高速的计算机系统,人们越来越多地将目光集中到了定制硬件技术上。同时,FPGA的发展和普及大大降低了定制硬件的开发难度和成本。通过在FPGA上实现定制硬件加速器,将应用性能提升几十到几百倍的案例在学术界已经屡见不鲜。而苹果、微软、谷歌等大型IT企业,目前也已纷纷开始或计划将硬件加速技术应用到电子产品和服务器当中。在可预见的未来,具备软硬结合设计能力的工程师将会更加具有竞争力。

《CPU 自制入门》是为读者打开硬件设计大门的理想教材。通过阅读本书,软件工程师能够更加了解硬件与底层,开发出高效代码。硬件工程师则可以在本书基础上设计定制硬件,进而开发高性能计算机系统。相信所有读者都可以在本书的阅读过程中受益匪浅,零距离地体验自制计算机系统的乐趣。

赵谦 (@JonsonXP)

2013年11月

声明

本书以提供知识为目的,请在明确判断、自负责任的基础上运用本书。使用本书信息所产生的后果,出版社与作者们概不负责。

本书内容以著作(日文版)完成时间——2012年9月为准,在您阅读本书时,实际情况可能有所改变。

如果没有特别声明,本书所用软件的版本全部为 2012 年 9 月的版本。 这些软件如有升级,可能会出现与本书所述功能或界面不符的情况。购 买本书前,请务必确认软件的版本号。

请在接受以上声明的条件下阅读本书。如果您未阅读这些声明,就贸然向出版社或作者们咨询上述相关问题,我们不会答复。望周知,请多多包涵。

- Microsoft Windows 是美国 Microsoft Corporation 的注册商标。
- 另外,本文中记载的商品名、公司名等,皆为各相关企业的商标或注册商标。

作者序

本书从零开始设计 CPU,通过这一过程,旨在让读者理解 CPU 的内部构造,并向读者传递设计 CPU 的乐趣。

虽然本书的主要目标是 CPU 设计,但除了 CPU,我们还要设计控制相关设备的 I/O、总线等,实际上是 SoC 设计。本书不但会讲解 CPU 设计,还涉及电路板设计、软件设计等计算机系统的全部要素。从硬件到软件,我们要全部从零开始设计、制作,最终上机调试。通过将 CPU设计、电路板设计以及软件设计的知识系统地整理到一本书中,我们可以更深入地了解计算机体系的各部分以及它们的关联。

本书的自制 CPU 是在 FPGA 上实现的。近年来,高性能 FPGA 的价格越来越便宜,个人用户也可以充分体验 FPGA 的乐趣。设计过程中,我们使用免费工具软件,挑选读者方便购买的零件,极力降低制作成本。

CPU、I/O 以及总线等相关 HDL 代码和软件程序代码都可以从技术评论社(http://gihyo.jp)的本书支持页面下载。不过,主板我们不随书赠送,而是给出成品供您参考。这样读者就可以根据自己的兴趣,制作自己想做的部分。

本书的目标读者主要是志在成为工程师的学生,因此,我们尽量减少阅读时所需背景知识,降低难度,以便更多人可以阅读。本书与其他技术书籍的最大不同在于,我们更强调动手实践,激发读者动手制作的乐趣。从使用 FPGA 设计、制作 CPU 到制作电路板以及开发软件,这些全部都能亲自动手实现。这是本书的主旨所在。比起在 PC 上编一点实验小程序,简单地在杂志附送的单片机上运行,本书的实践更让人有成就感。

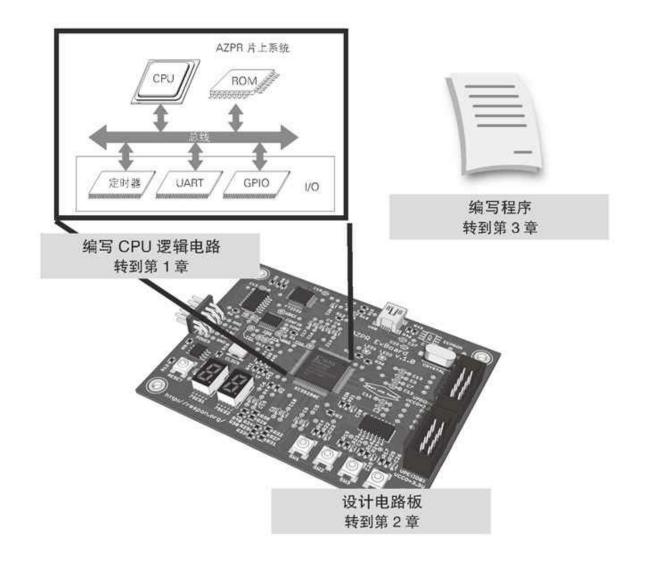
本书虽极力减少阅读所需的背景知识,但逻辑代数、编程语言、计算机 架构等基础知识还是要必备的。关于这些知识,本书虽然会做些介绍, 但因篇幅所限,无法一一系统讲解。本书主要着眼于"动手制作",基础 知识讲解不到位之处敬请谅解。我们也会在专栏部分介绍一些书籍,它 们有助于理解本书的背景知识。 本书适合大学、大专院校信息、电子专业的学生阅读。将来想学习这类专业的高中生或者对计算机感兴趣的读者都可以阅读。虽然本书算不上一看就懂,但只要带着兴趣阅读就可以充分理解。

2012年9月

本书的阅读方法

本书分为3章。第1章以介绍 CPU 为主,同时介绍如何制作存储程序与数据的内存、与外部进行输入输出的I/O 以及将这些模块连接起来的总线,这些模块可以组合成一个简单的计算机系统。第2章进行电路板的设计和制作,好让这个计算机系统运转起来。在第3章中,我们为这个计算机系统编写程序,并上机测试。本书最大的特点是,可以自己制作整个计算机系统。

这3章彼此独立,读者可以根据自己的兴趣选择阅读。



下面详细介绍本书这 3 章。第 1 章为 CPU 逻辑设计,第 2 章为电路板设计,第 3 章为软件设计。



第1章的 CPU 设计中,要设计 CPU、内存、I/O 以及连接这些模块的总线,我们使用硬件描述语言 Verilog 实现,最终将这些模块组合形成简单的计算机系统。我们首先讲解计算机、数字电路、Verilog HDL 的基础。然后按照总线、内存、CPU、I/O 的顺序制作计算机。另外,还会介绍 Verilog HDL 的仿真环境。

第2章的电路板设计是为了让我们能在实际的硬件上调试制作的 CPU 与程序。我们使用一种叫 FPGA 的芯片来制作 CPU,它的特点是可以对其内部构造进行编程重构。大体制作流程为挑选必要的元件、制作电路图和布局图,然后制作印刷电路板。电路板制作部分我们会介绍感光电路板制作法和外包给制板公司制造两种方法。最后将元件组装到制作完成的电路板上,进行功能检查。

第3章的软件设计中,我们为所设计的 CPU 开发程序,并在做好的电路板上调试。首先对开发环境进行说明,介绍所需的开发工具以及各个工具的安装、使用方法,然后讲解编程。我们运用实例程序讲解 CPU、I/O 的使用方法,并在做好的电路板上运行程序。



本书的最终成果是在实际的电路板上运行演示程序。本书的重点不是"可以做什么",而是"亲手制作",因此,并没有设计很复杂的演示程序。如果只是想实现复杂功能,使用市面上销售的单片机更容易一些。但是从自己动手制作计算机这方面讲,仅仅在单片机上运行程序是无法获得这种满足感的。对于正在使用单片机电路板进行电子制作的读者来说,阅读本书后一定可以更深入地理解逻辑设计、电路板设计和程序设计。我们经常会遇到使用现成通用元件无法实现的功能,届时再回顾一下本书一定会对你有所帮助。

第1章 CPU的设计与实现

本章中,我们首先着手设计 CPU、内存、I/O 以及它们之间的连接总线,随后使用硬件描述语言 Verilog HDL 进行实现。最终将这些模块组合,形成一台简单的计算机。

本章最大的特点是使用硬件描述语言实现计算机的各个基础部件,并详 细讲解制作过程。通过学习本章内容,我们不仅可以理解计算机的各组 成要素,还能动手制作并实现它们。

- 1.1 序
- 1.2 计算机系统
- 1.3 数字电路基础
- 1.4 Verilog HDL 语言
- 1.5 系统蓝图
- 1.6 总线的设计与实现
- 1.7 存储器的设计与实现
- 1.8 AZ Processor 的设计与实现
- 1.9 I/O 的设计与实现
- 1.10 AZPR SoC 整体连接
- 1.11 AZPR SoC 的仿真
- 1.12 本章总结

1.1 序

本章将实现一台简单的计算机系统的 SoC(System-on-a-Chip,片上系统)。它以 CPU 为核心,同时实现了负责存储程序和数据的内存、负责和外部进行输入输出的 I/O 以及它们之间的连接总线。SoC 是将一整套系统集成到单一芯片的集成电路设计方法。

开发之前,我们先来确定 CPU 的名字。我们为这次开发的 CPU 取名为 AZ Processor,因为本书旨在从头到尾亲自动手设计和实现一台计算机,这几个英文字母就含有从 A 到 Z 全部亲手制作的意思。然后,AZ Processor、内存、各种 I/O 通过总线连接形成的 SoC,我们称之为 AZPR SoC(AZ Processor 片上系统)。图 1-1 为 AZPR SoC 的概要。

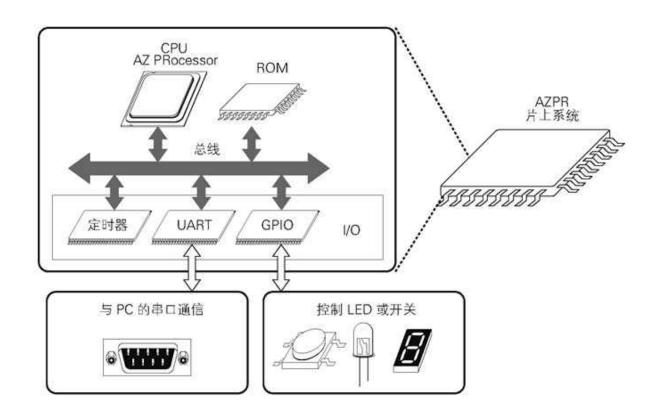


图 1-1 AZPR SoC 的概要

图 1-2 列出了本章的结构。1.2 节~1.4 节分别简单介绍计算机系统、数字电路基础和 Verilog HDL 语言。这 3 节的内容是制作 AZPR SoC 需要

掌握的最基础的知识。已经掌握这些知识和设计经验的读者,可以跳过此部分。

1.5 节~1.10 节是本章主要的设计和实现部分。1.5 节将对 AZPR SoC 进行说明。1.6 节~1.9 节将分别对总线、内存、CPU 和 I/O 的设计和实现进行说明。1.10 节将各个模块连接,完成 AZPR SoC 的制作。1.11 节介绍 AZPR SoC 的仿真。最后的 1.12 节对本章进行总结。

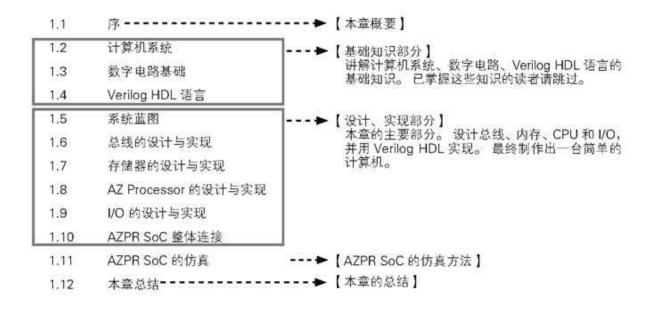


图 1-2 本章的构成

1.2 计算机系统

本节将介绍计算机系统的构成要素及其功能。

1.2.1 什么是计算机

计算机是根据程序进行运算和数据处理的计算机器。近年来,随着 PC(Personal Computer,个人电脑)在普通家庭中的广泛普及,计算机 对我们的生活产生了深远的影响。如今,不仅是 PC,与我们生活息息 相关的手机、家电等也广泛应用了计算机。

通常,计算机由以下几部分组成:负责计算和处理数据的 CPU、负责存储程序和数据的存储器,以及和外部进行数据交换的 I/O(Input/Output,输入输出装置)。各部分通过总线连接就构成了一台计算机。

计算机的构成要素如图 1-3 所示。以 PC 机的组成为例,一般使用 Intel 或 AMD 公司的 CPU,DDR3 SDRAM 之类的内存,另外还有键盘、鼠标、显示器等 I/O。这些 CPU、内存、I/O、总线并不局限于 PC,多数计算机都是由这四大要素组成。

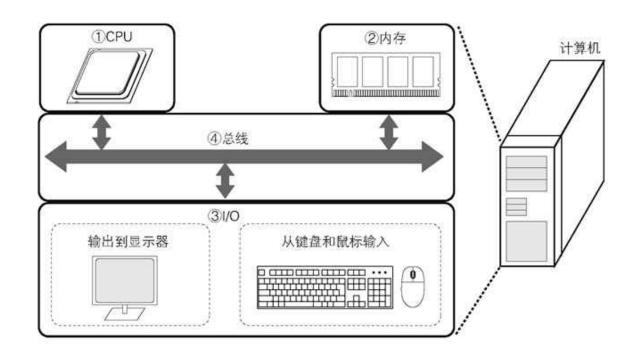


图 1-3 计算机的构成要素

1.2.2 什么是 CPU

CPU 是计算机中进行各种运算和数据处理的装置。CPU 是 Central Processing Unit(中央处理器)首字母的缩写。近年来,商用 CPU 基本都基于集成电路技术制造,然后封装到图 1-4 所示的包装后出售。

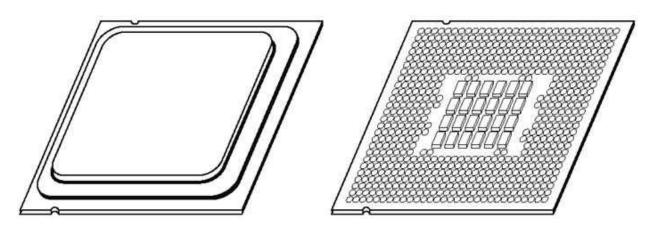


图 1-4 CPU 的外观

CPU 是一种根据指令进行各种处理的电子电路。图 1-5 展示的是 CPU 的处理流程。内存存储着可由 CPU 执行的指令集合所组成的程序。CPU ①读取(Fetch)内存中的指令,然后对其要处理的操作进行②解码(Decode),最后进行③执行。

CPU 基本上就是在这三种状态之间进行任务处理。这种将存储在内存中的程序读出再执行的架构称为存储程序式架构 1 。

 1 这种架构的计算机被称为存储程序计算机(Stored-program computer)。——译者注

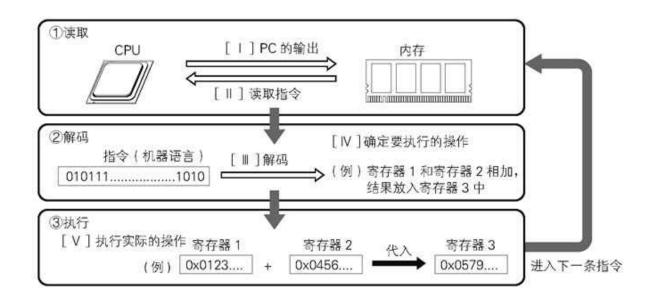


图 1-5 CPU 的处理流程

• ①读取

首先,CPU 要把即将执行的指令从内存中读取出来。CPU 中有个PC(Program Counter,程序计数器)寄存器,其中保存着即将执行的指令的地址。指令的读取是通过将 PC 寄存器的值输出给内存,由内存返回该值对应地址中的指令。

• ②解码

然后,CPU 对读取的指令所对应的操作进行解码。指令有很多种,有进行各种运算的指令、控制下一条命令的指令、对内存和 I/O 进行读写的指令,还有对 CPU 进行控制的指令。这些指令由 CPU 中

被称为指令解码器的模块进行解码。可以用来保存地址和运算结果的寄存器称为通用寄存器(General Purpose Register)。

• ③执行

最后,CPU 对解码器确定的操作进行处理。CPU 可以从内部存储 装置——寄存器或外部的内存读取数据并处理,然后将结果写回寄 存器或内存。

简化的 CPU 内部构造图如图 1-6 所示。读取指令时,CPU 将 PC 寄存器的值输出到内存,然后从内存中将对应的指令取回。取回的指令保存在指令寄存器中。指令解码是将储存在指令寄存器的指令解码,确定将要处理的操作。大多数情况下,在确定即将处理的操作的同时,CPU 会从通用寄存器中读取运算要使用的数据。指令执行时,从通用寄存器将操作数值取出,通过运算器处理然后将结果写回。CPU 执行的运算结果可以写回通用寄存器,也可写入内存。CPU 也可以从内存读取数据作为结果返回。

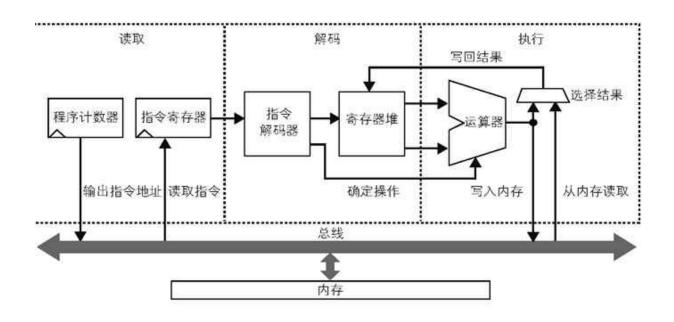


图 1-6 CPU 的内部构造

CPU 执行的指令,由代表操作种类的操作码和代表操作对象的操作数两部分组成。指令的构造如图 1-7 所示。指令本身用特定的二进制序列来表示,这种二进制序列称为机器语言。

机器语言的二进制序列 100111000101101010110100100000

图 1-7 指令的构造

操作数是由寄存器地址、内存地址或立即数来指定的。立即数是指嵌入指令中的固定常数。操作数的数量和位宽根据 CPU 和指令的不同而不同。根据可使用的操作数的数量,指令可以分为 3 操作数形式、2 操作数形式和 1 操作数形式等。

根据执行的指令的特征,CPU 分为 RISC(Reduced Instruction Set Computer,精简指令集计算机)和 CISC(Complex Instruction Set Computer,复杂指令集计算机)两种。表 1-1 比较了 RISC 和 CISC 的特征,并给出了其代表产品。

表 1-1 RISC 和 CISC 的比较

	指令功能	指令数量	硬件	高速化	执行相同处理时 的指令数	代表产品
RISC	单纯	少	简单	适合	多	IBM Power、Sun MicroSystems SPARC、MIPS、ARM 等
CISC	复杂	多	复杂	不适合	少	Intel i386、IBM System/360、DEC PDP 等

RISC 类 CPU 的指令功能单纯,种类较少。相对应地,CISC 类 CPU 的指令功能复杂,种类繁多。RISC 指令精简的好处是 CPU 内部构造可以简化,适合高速操作。但是在进行相同操作时,由于每一条指令都功能单纯,所以与 CISC 相比,它需要使用更多的指令数量。虽然 CISC 的内部构造复杂不适合高速操作,但进行相同处理时指令数比 RISC 要少。

RISC 架构最大的特点是只使用载入和存储指令访问内存,这种架构称

为载入存储架构(Load/Store Architecture)。这样做的好处是可以简化 指令集和流水线设计。在这种架构下,运算指令只能对寄存器中的数据 进行操作。

RISC 和 CISC 两种架构各有所长,孰优孰劣不能一概而论。在追求高速运作的 CPU 的领域中,RISC 被认为更具优势。这些年,虽然 Intel 和 AMD 两家公司的 CPU 指令集依然是 CISC 的,但内部却将复杂指令分解为简单指令,使得内部可以像 RISC 一样工作。

专栏

CPU 的位宽

CPU 的位宽表现了 CPU 可以访问的地址空间或数据的大小。比如,32 位 CPU 可以处理 32 位的数据,可以访问的地址空间为 4G字节(2 的 32 次方)。随着程序、数据的规模和内存容量的增大,32 位 CPU 有些不太够用,最近的 CPU 一般都是 64 位。 CPU 的位宽并没有明确的定义。有根据寄存器或地址的宽度划分的,也有根据指令或总线宽度划分的各种标准。现在大家普遍将 CPU 可以处理的整数型数据的宽度定为位宽。实际上,根据 CPU 厂家的想法和主张,解释也不尽相同。除了位宽,CPU 可以访问的地址空间或数据的大小还用字(word)来表示。通常,CPU 的字长和位宽是一致的。

1.2.3 什么是内存

内存是用来存放运行时指令(程序)和数据的存储器。为了和计算机中长期保存数据和程序的存储器区别,内存有时也称为主存(Main memory)。

最近的计算机通常采用 DRAM(Dynamic Random Access Memory,动态随机存储器)技术的内存。DRAM 是通过在电容器中积蓄电荷来保存数据的存储元件。电容器中充电状态是 1,放电状态是 0,以此来表示数值。由于电容器中的电荷一段时间后会衰减,所以 DRAM 需要定期进行重新写入数据的刷新(Refresh)操作。根据访问方式和规格的不同,DRAM 分为 SDRAM(Synchronous DRAM,同步 DRAM)和 DDR SDRAM(Double Data Rate SDRAM,双倍数据率 SDRAM)等种类。

内存使用地址的概念来管理存储的数据。地址表示的是数据存储的位置,如同数据的住所一样。每个数据单元都有一个地址。大多情况下数据单元是一个字节(8位)长度。这种方式称为字节编址。图 1-8 说明了内存和地址的关系。

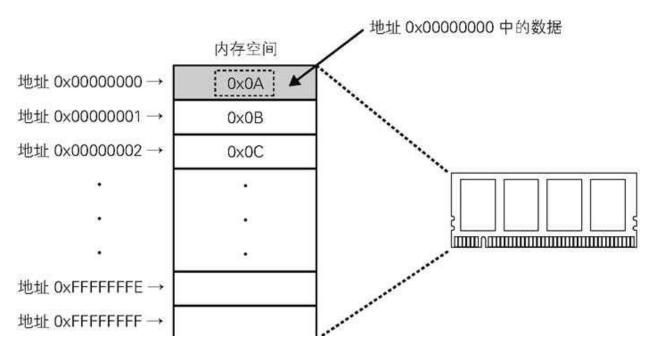


图 1-8 内存和地址

内存等存储器的特点是速度越快成本越高。因此通常使用"高速小容量"、"中速中等容量"到"低速大容量"等多种存储器组合的混合型架构。这种构造称为存储器层级。图 1-9 是存储器层级的示例。

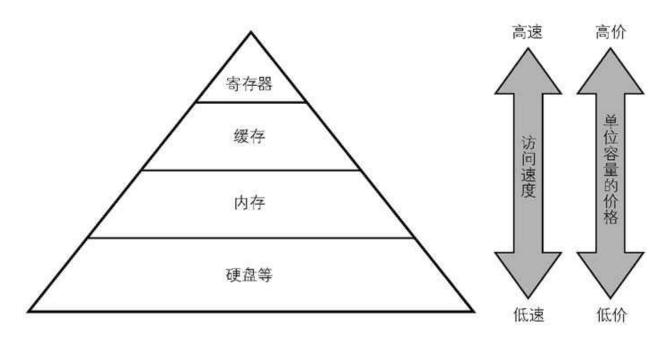


图 1-9 存储器层级示例

在存储层面,速度最快的是 CPU 中的寄存器。CPU 比内存速度快很多,由 CPU 直接访问内存效率较低。为了提高内存访问速度,在 CPU 和内存间增加了被称为缓存的高速小容量存储器。

缓存可以暂时性地缓冲存储从内存中读取的数据。CPU 在访问内存时,如果需要的数据已经保存在缓存中,则可直接从缓存中读取,以提高访问效率。根据容量和速度的不同,缓存也分为多个层级,通常为一级缓存、二级缓存等多个级别。

1.2.4 什么是 I/O

I/O(Input/Output)是进行数据输入输出的装置。计算机通过 I/O 和外部实现数据交换。计算机的处理操作按照从外部读取数据、在内部处理数据、再向外部输出结果的顺序进行。以个人电脑为例,如图 1-11 所示,它从鼠标或键盘输入数据,处理器根据程序处理数据,通过显示器等向外部输出结果。

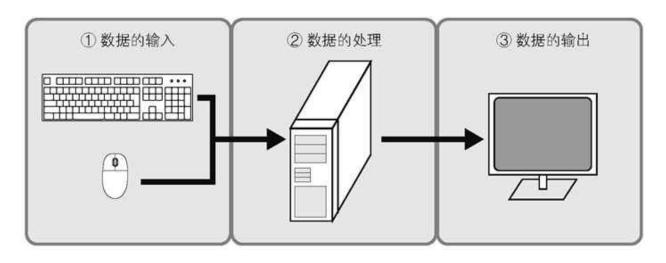


图 1-11 计算机的处理流程

专栏

字节序

将多字节数据存储在内存中时,各字节的存储顺序称为字节序。比如,将4字节数据0x12345678放入内存时,地址0中放0x12、地址1中放0x34、地址2中放0x56、地址3中放0x78的方式,称为大端序。相对地,地址0中放0x78、地址1中放0x56、地址2中放0x34、地址3中放0x12的方式,称为小端序。这两种数据存储方式请参见图1-10。

对人类来说,大端序理解起来比较容易,然而对计算机来说,小端比较容易操作,因为不同长度数据的低位位置是相同的。

不同的 CPU 采用的字节序也不尽相同,由此产生的软件通用性和可移植性的问题也屡屡发生。Intel 公司的 x86 架构采用的是小端序,而 Sun(现属 Oracle)公司的 SPARC 处理器和 MIPS 科技公司的 MIPS 处理器等采用的是大端序。

最近,很多处理器考虑到软件的通用性和可移植性,同时支持两种字节序并可依据程序切换,这种方式称为双端序。

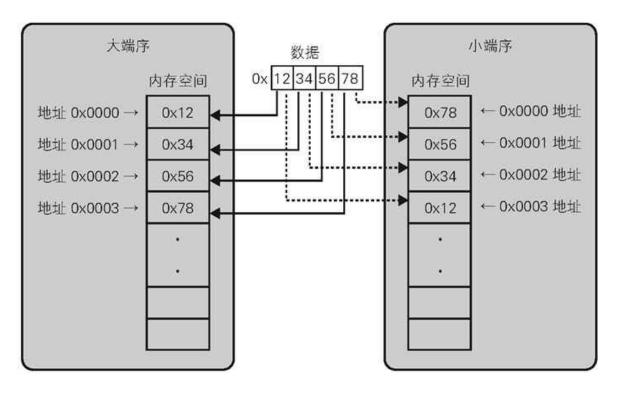


图 1-10 字节序

访问 I/O 的方式大致分为存储器映射 I/O 和端口映射 I/O 两种。

存储器映射 I/O 方式中,I/O 也和内存一样使用地址进行管理,可以和访问内存一样的方式进行访问。存储器映射 I/O 的概要如图 1-12 所示。存储器映射 I/O 方式中,由于使用访问内存的指令进行 I/O 访问,硬件上较为简化。但缺点是,I/O 也会占用地址 空间。

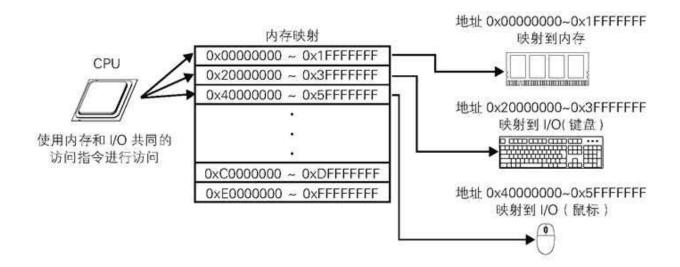


图 1-12 存储器映射 I/O

端口映射 I/O 方式中,CPU 含有支持访问 I/O 的专用指令。端口映射 I/O 的概要如图 1-13 所示。端口映射 I/O 方式的优点,一是地址空间可以全部分配到内存,二是内存和 I/O 的访问可以在指令级别区分。但是,由于需要专用指令,缺点是硬件设计变得复杂。

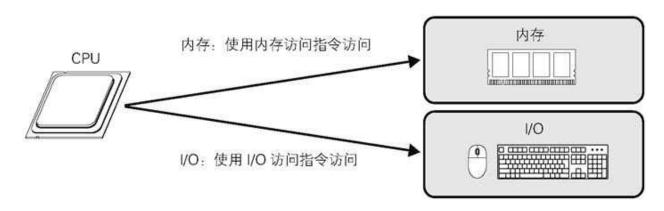


图 1-13 端口映射 I/O

1.2.5 什么是总线

总线是 CPU、内存和 I/O 之间交换数据的共同通道。总线将一根信号线在多个模块间共享进行通信。图 1-14 是总线的示例。

两个模块通过总线交换数据时,发起访问的一侧称为总线主控,接受访问的一侧称为总线从属。图 1-14 的示例中,CPU 为总线主控,内存、I/O 等为总线从属。

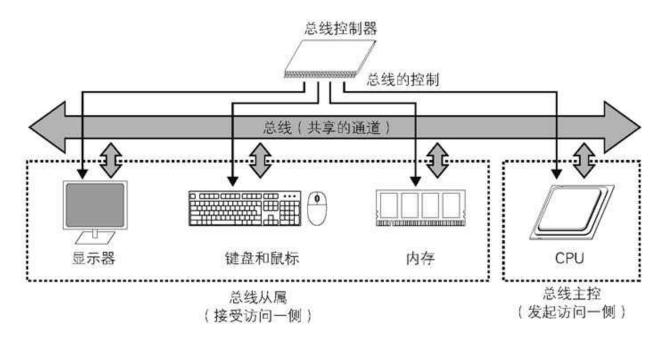


图 1-14 总线示例

总线一般由数据总线、地址总线和控制总线构成。数据总线用来传输交换的数据,地址总线用来指定访问的地址,控制总线负责总线访问的控制。各个信号的时序、进行交换的规则等称为总线协议。通过总线交换数据的整个过程称为总线传输。总线传输的示例请参见图 1-15。

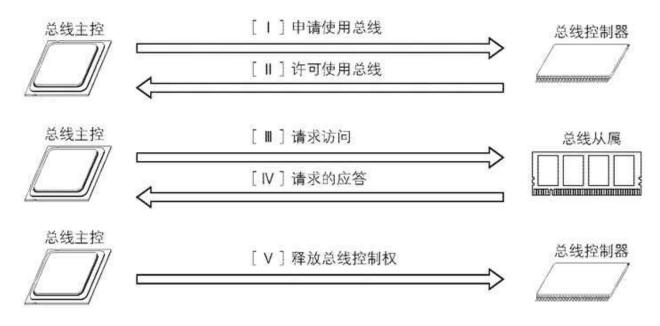


图 1-15 总线传输示例

[I] 申请使用总线

多数情况下,总线上接有多个总线主控,由于总线是共享的通道,不能同时使用多个总线主控。因此,需要对多个总线主控的使用请求进行调停。访问总线的权力称为总线控制权,对多个访问的调停称为总线仲裁。总线仲裁由总线控制器内的仲裁器实施。总线主控在访问总线之前先向总线控制器申请总线控制权。

[II] 许可使用总线

总线控制器对多个总线主控的请求进行调停,依据仲裁规则对总线的使用进行许可授权。

[III] 请求访问

获取总线控制权的总线主控对总线从属发送访问请求。请求中包含"要访问哪个地址"、"是读取访问还是写入访问"和"写入时的数据"等信息。

由于总线是共享的通道,总线主控输出的信号会发送到所有总线从属。 因此使用片选信号(Chip select,芯片选择信号)等控制信号来区别对 哪个从属进行访问。每个总线从属都设有片选信号,可以使用片选信号 选择要访问的总线从属。

一般的总线结构会为每个总线从属分配地址空间。总线控制器内的地址解码器根据要访问的地址产生片选信号。

[IV] 请求的应答

接受访问的总线从属会根据请求对总线主控进行应答。针对请求,应答时采用 Ready 等控制信号。在接受读取请求时,应答的同时输出读出的数据。

[V]释放总线控制权

总线使用完毕,总线主控通知总线控制器释放总线控制权。

专栏

总线的优缺点

总线的优点是只要遵循总线协议,任何设备都可以简单地进行连接。并且由于使用的是共享通道,硬件的成本也比较低。但是,数据传输的吞吐量较低。

近几年,一台计算机搭载多个 CPU 的情况比较常见。随着与总线通信的 CPU 数量的增多,总线很容易变得拥堵。因此,业内也在开发各个节点通过网络连接的技术来替代传统的通道共享的总线。

1.2.6 小结

本节介绍了计算机的基本概念。多数计算机是由 CPU、内存、I/O 以及连接它们的总线构成。计算机是通过 CPU 将存储在内存的指令读出并执行、通过 I/O 进行数据的输入输出来实现处理的。

专栏

计算机相关书籍

每节节末的专栏会介绍和该节相关的书籍。这些书籍有助于读者更全面、系统地理解该节的知识。

• コンピュータはなぜ動くのか(矢沢久雄著、日経 **BP** 社) (中文译名《计算机为何能工作》)

这本书详细介绍了计算机基本知识,涉及硬件、软件、编程、网络等各方面的内容,可以帮助读者理解计算机及其相关技术。这本书并非专业图书,非计算机专业的读者也很容易阅读。

• 構造化コンピュータ構成(Andrew S. Tanenbaum 著、長尾 高弘訳)

(原书名 *Structured Computer Organization* ,中文译名《计算机组成:结构化方法》)

这本书可以作为大学、大专院校计算机科学专业学生的教材,帮助读者系统地学习计算机相关知识。原著作者 Tanenbaum 曾编写过多本优秀的教科书。笔者在此将本书推荐给想真正学

好计算机的读者。

1.3 数字电路基础

本节将介绍数字电路的基础知识。数字电路是利用数字信号的电子电路。近年来,绝大多数的计算机都是基于数字电路实现的。

1.3.1 什么是数字电路

数字电路是利用两种不连续的电位来表示信息的电子电路。数字电路中的电源电压 H(High,高)电平、接地电压 L(Low,低)电平分别代表 1和 0,以此实现信息的表达。大部分数字电路是基于叫做 MOSFET(Metal-Oxide-Semiconductor Field-Effect Transistor,金属氧化物半导体场效应管)的场效应管实现的。在数字电路中,MOSFET 通过组合可以实现各种各样的逻辑电路。

1.3.2 数值表达

数字电路中的信息由 0 和 1 两个数字表示,因此数字电路的设计基于二进制数(binary number)。二进制是指从 0 到 1 的数值在一位数字中表示,遇 2 则向上进位的数值表达方式。二进制的第 n 个数字位,数值上是 2 的 n - 1 次方位。我们平时使用的数值表达方式是十进制(decimal number),十进制中,0 到 9 的数值可在一位中表示。图 1- 16 说明了二进制和十进制的位值关系。

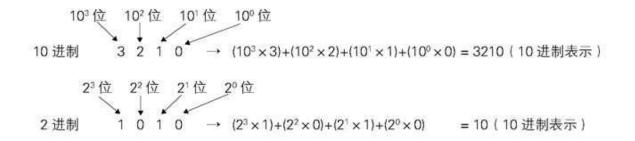


图 1-16 二进制和十进制的位值关系

十进制数的 3210 可以表示为($10^3 \times 3$)+($10^2 \times 2$)+($10^1 \times 1$)+($10^0 \times 0$)。二进制数的 1010 可以表示为($2^3 \times 1$)+($2^2 \times 0$)+($2^1 \times 1$)+($2^0 \times 0$),相当于十进制数 10。一个数字位上可以表达数值的个

数称为底数,十进制的底数是10,二进制的底数是2。

计算机中常用的数值表现方式,除了二进制和十进制之外,还有八进制(octal number)和十六进制(hexadecimal number)等。八进制使用从0开始的八个数表达数值。十六进制中,从10到15使用字母A到F来表示,以0到9加上A到F表示十六个数值。

八进制数值通常以 0 开头,以区分十进制等表达方式。十六进制则通常以 0x 开头。0x 中的 x 代表 hexadecimal 中的 x。十六进制也有在末尾加 H 等其他表达方法。

表 1-2 列出了利用以上几种进制表达数值的例子。

表 1-2 数值表现的示例

10 进制数	2 进制数	8 进制数	16 进制数
0	0000	000	0x0
1	0001	001	0x1
2	0010	002	0x2
3	0011	003	0x3
4	0100	004	0x4
5	0101	005	0x5
6	0110	006	0x6
7	0111	007	0x7

8	1000	010	0x8
9	1001	011	0x9
10	1010	012	0xA
11	1011	013	0xB
12	1100	014	0xC
13	1101	015	0xD
14	1110	016	0xE
15	1111	017	0xF

1.3.3 有符号二进制数

在用二进制表示有符号数值时,我们经常使用补码表示法。补码表示法中,N位的二进制数的最高位代表数值 - (2^{N-1}) 。图 1-17 介绍了有符号二进制数的表达方式。

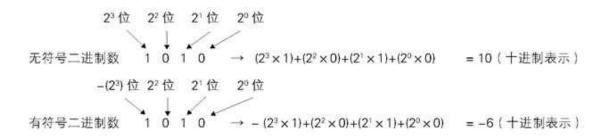


图 1-17 有符号二进制数的示例

专栏

比特和字节

二进制中的一个数字位称为 binary digit, 简称比特(bit)。计算机领域中,我们使用比特作为单位来表示数据量,还会用到一种叫字节(byte)的单位。通常一个字节代表 8 比特,绝大多数 CPU 都是以字节为单位处理数据的。内存地址大多也是为每字节赋予一个地址,称为字节编址方式。由 8 比特组成一个字节是出于 2 的 8 次方表达的范围(0~255)比较适合表达文字(英文字母、符号、控制符等)的考虑。

专栏

1K 字节有多大

K、M、G、T 是表示大数据量时常用的单位。1K 的大小有 1000 (10 的 3 次方) 和 1024 (2 的 10 次方) 两种计数方法。

通常,衡量计算机内存和网络数据包大小时,1K 相当于 1024 比特。而在硬盘等存储器的标签上记述的尺寸或物理学中的1K 相当于1000。

表 1-3 是对单位的说明。	位的说明。	是对单	1-3	表
----------------	-------	-----	-----	---

	1K 等于 1024 时	1K 等于 1000 时
1 [K]	1 024 (2 的 10 次方)	1 000 (10 的 3 次方)
1 [M]	1 048 576 (2 的 20 次方)	1 000 000 (10 的 6 次方)
1 [G]	1 073 741 824 (2 的 30 次方)	1 000 000 000 (10 的 9 次方)
1 [T]	1 099 511 627 776 (2 的 40 次方)	1 000 000 000 000 (10 的 12 次 方)

无符号二进制数变成补码时,将所有比特反转(又称取反码)后加 1。 以 4 位二进制数 0001 为例,全比特反转后为 1110,然后加 1 成为 1111。也就是说,在二进制的补码表示法中,将数字 1 表示为 0001,-1 表示为 1111。这就是说最高位的比特起到了符号位的作用。最高位为 0 时是正数,最高位为 1 时是负数。 二进制补码表示法的好处是正数和负数相加时无需考虑符号的处理。以刚才例子中的 1 和 -1 的补码相加为例,0001 加 1111 后进位得到10000。当数据宽度为 4 位时忽略第五位的 1,结果为 0000,也就是正确答案——数值 0。如上所示,运用二进制补码表示法可以在不关心数据符号的情况下进行运算。

1.3.4 MOSFET 的结构

近年来,数字电路基本上都是由 MOSFET 场效应管构成的。MOSFET 是一种在施加电压后可以像开关一样工作的半导体器件。MOSFET 有 P型 MOSFET 和 N型 MOSFET 两种。P型 MOSFET 的构造如图 1-18 所示,N型 MOSFET 的构造如图 1-19 所示。

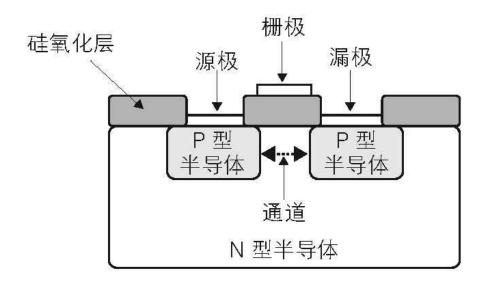


图 1-18 P型 MOSFET 的构造

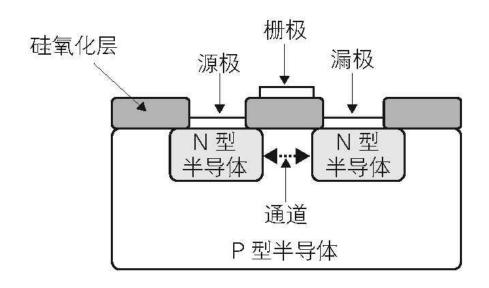


图 1-19 N型 MOSFET 的构造

MOSFET 有源极、漏极和栅极 3 个电极。功能上,源极、漏极和栅极分别作为电流输入、电流输出和电流控制使用。MOSFET 的源极和漏极采用相同类型的半导体材料,而栅极下的通道则填入不同类型半导体材料。P型 MOSFET 的源极和漏极使用 P型半导体,栅极下的通道使用 N型半导体。N型 MOSFET 材料的构成与 P型 MOSFET 相反。

下面以N型MOSFET为例说明其工作原理。在不给控制电流的栅极施加电压时,源极和漏极间填充了异种半导体材料,因此电流无法流过。当给栅极施加正电压时,源极和漏极中N型半导体材料里的自由电子被栅极吸引,使通道中充满电子,源极和漏极间的电流从而能够流动。

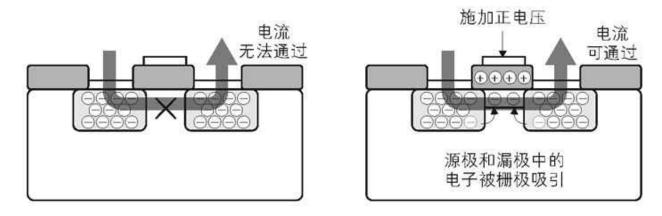


图 **1-20** N型 **MOSFET** 的动作原理

N型 MOSFET 在栅极施加电源电压(H)时电流可以流通,接地(L)

时电流无法流通。反之,P型 MOSFET 的栅极接地时电流可以通过,施加电源电压时电流无法流过。这种持有相反特性的 N型 MOSFET 和P型 MOSFET 互补使用形成的门电路称为 CMOS(Complementary Metal Oxide Semiconductor,互补金属氧化物半导体)。CMOS 可以用来制作各种各样的逻辑电路。

1.3.5 逻辑运算

逻辑运算是只用"真"、"假"二值进行的运算。数字电路中的 H(1) 和 L(0) 可与逻辑运算中的"真"、"假"对应,进行逻辑运算。逻辑运算使用 AND(逻辑与)、OR(逻辑或)、NOT(逻辑非)三种基本运算组合 来实现各种运算。图 1-21 对基本的逻辑运算进行了说明。

逻辑		真值表		文氏图
	A	В	Y	
1110	0	0	0	
AND	0	1	0	(A () B
	1	0	0	
	1	1	1	
OR	A 0 0 1 1 1	B 0 1 0	Y 0 1 1 1 1	A B
NOT	0 1		Y 1 0	A

图 1-21 基本逻辑运算

图 1-21 中 A 和 B 为输入, Y 为运算结果。AND 运算在输入 A 和 B 双

方都为真时结果Y为真,其他情况下Y为假。因此AND运算的结果是A和B的交集。

OR 运算在输入A和B任意一方为真时结果Y为真,A和B双方皆为假时结果Y为假。因此OR运算的结果是A和B的并集。

NOT 运算是单输入的运算,输入为真时结果为假,输入为假时结果为真。因此 NOT 运算的结果是输入 A 的补集。

1.3.6 CMOS 基本逻辑门电路

接下来介绍 CMOS 的基本逻辑门电路。N型 MOSFET 和 P型 MOSFET 的电路符号 如图 1-22 所示。



图 1-22 MOSFET 的电路符号

将 MOSFET 按照图 1-23 的方式组合即可实现 NOT 门电路。当输入 H 时,N 型 MOSFET 打开,输出为 L; 当输入 L 时,P 型 MOSFET 打开,输出为 H。

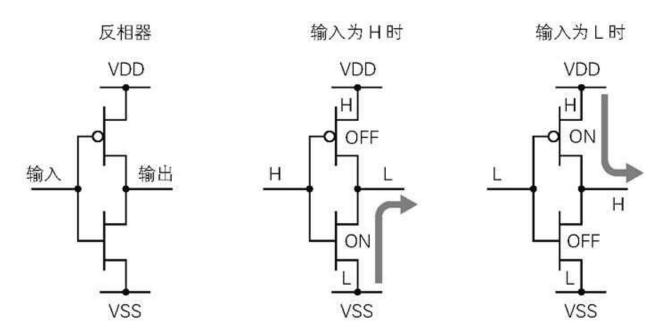


图 1-23 NOT 门电路的电路图和动作原理

从最简单的 NOT 门电路到各种逻辑门电路,都可以由 MOSFET 的组合进行实现。图 1-24 中列出的是逻辑门电路中定义的基本逻辑门电路。逻辑门电路的电路符号称为 MIL(美军标准)逻辑符号。数字电子电路通过基本逻辑电路的组合来实现各种逻辑电路功能。

逻辑	真值表	MIL 逻辑符号	逻辑	真值表	MIL 逻辑符号
与	A B Y 0 0 0 0 1 0 1 0 1 1 1 1	A Y	与非	A B Y 0 0 1 0 1 1 1 0 1 1 1 0	A Y
或	A B Y 0 0 0 0 0 1 1 1 1 1 1 1 1	A B	或非	A B Y 0 0 1 0 1 0 1 0 0 1 1 0 0 1 1 0 0	A Y
异或	A B Y 0 0 0 0 1 1 1 0 1 1 1 0	A Y	异或非	A B Y 0 0 1 0 1 0 1 0 0 1 1 1 1	A Y
非	A Y 0 1 1 0	A Y	1		

图 1-24 基本逻辑门电路

1.3.7 存储元件

通过组合基本的逻辑门,可以实现用来保存数据的存储元件。锁存器(Latch)就是其中一种存储元件。锁存器具有像闩锁一样锁住并维持数据的特性。

图 1-25 是一种最为单纯的锁存器,其电路由一个 2 输入的 AND 门构成,并将输出与其中一个输入相接形成一条循环回路。一旦这个电路的输入 A 为 0 时,循环回路中的值就一直为 0。这样就可以利用循环回路将逻辑值锁存。

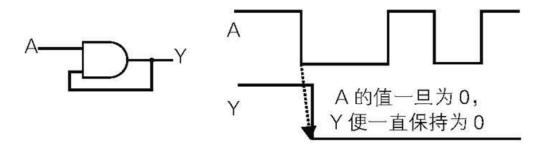


图 1-25 最简单的锁存器

还有一种锁存器叫 D 锁存器(Data Latch,D-Latch,数据锁存器)。D 锁存器的电路构造如图 1-26 所示,它由 4 个 NAND 门电路构成。D 锁存器中有 D (Data) 和 E (Enable)两个输入信号,Q 和 \overline{Q} 两个输出信号。D 锁存器在 E 为 0 时保持前一个数据,E 为 1 时将输入 D 的数据输出到 \overline{Q} 。 \overline{Q} 是输出信号 \overline{Q} 的反相信号。D 锁存器的真值表如图 1-27 所示。由于 D 锁存器在 E 为 1 时输入的 D 直接通过 \overline{Q} 输出,所以也称为通过型锁存器。

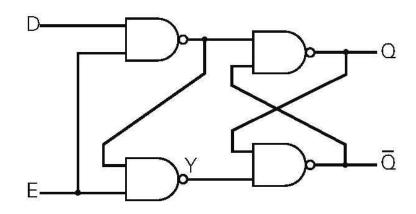


图 1-26 D 锁存器的构成及其电路符号

箱	输入		出	电路
E	D	Q	ā	电路
0	0	维持前一个 数据	维持前一个 数据	#持前一个数据 X a E の X a
0	3	维持前一个 数据	维持前一个 数据	□
1	0	0	1	線出信号 D 的值 下 1
1	1	1	0	输出信号 D 的值 D 0 0 0 0

图 1-27 D 锁存器的真值表

D 锁存器和 NOT 门组合,可以实现依据时钟信号同步并保存数据的 D 触发器。D 触发器的电路构成和符号分别如图 1-28 和图 1-29 所示。

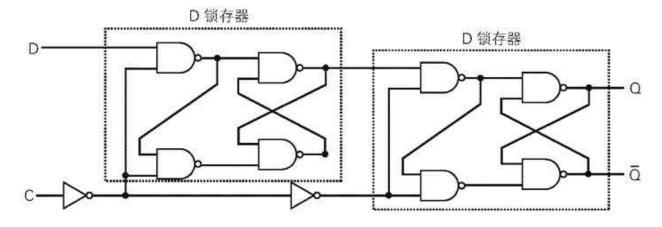


图 1-28 D 触发器的电路构成

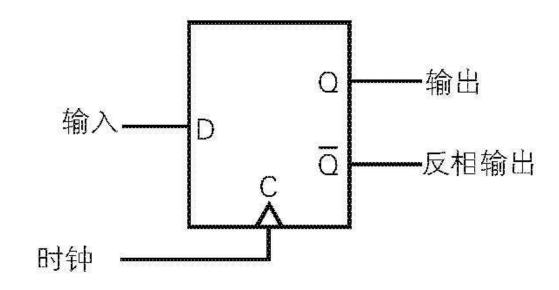


图 1-29 D 触发器的电路符号

D触发器有 D(Data)和 C(Clock)两个输入信号,Q和 \overline{Q} 两个输出信号。当 D触 发器的 C 为 0 时,前端 D 锁存器输出信号 D 的值,后端 D 锁存器保持之前的数据。当 C 为 1 时,前端 D 锁存器保持之前的数据,后端 D 锁存器将前端 D 锁存器保持的数据直接通过 Q 输出。D 触发器的动作原理和波形图分别如图 1-30 和图 1-31 所示。

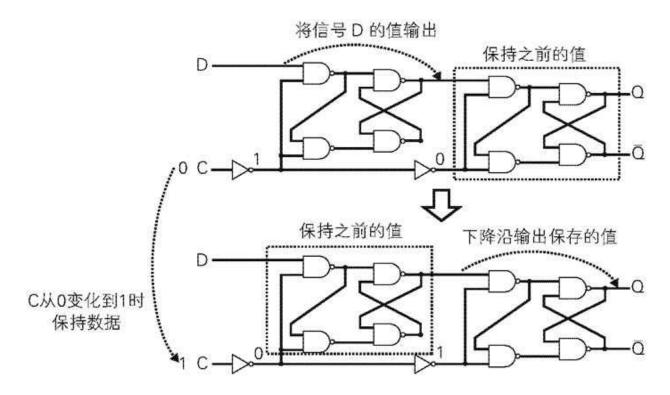


图 1-30 D 触发器的动作原理

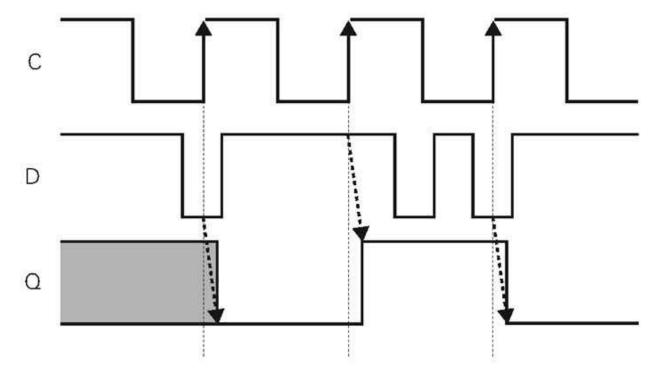


图 1-31 D 触发器的波形图

D 触发器由于原理简单,构造单纯,被广泛使用在同步电路当中。

专栏

建立时间与保持时间

D触发器是由时钟信号的边沿来触发数据的存储动作的。因此,需要在时钟沿前后一段时间内将输入信号稳定下来。如果在时钟变化时输入信号也在变化,很可能无法正确存储数据。因此,为了让 D触发器正确存储数据,需要有建立时间(setup time)和保持时间(hold time)两个基本条件。

建立时间是在时钟变化前必须稳定输入信号的时间,而保持时间是时钟变化后必须稳定输入信号的时间。

图 1-32 说明了建立时间和保持时间的关系。同时遵守建立时间和保持时间,就可以让 D 触发器正确的存储数据。

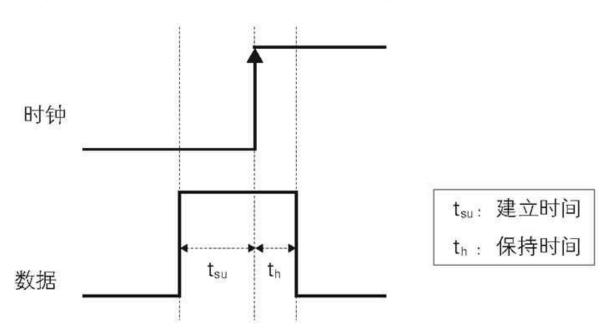


图 1-32 建立时间与保持时间

1.3.8 组合电路和时序电路

数字电路可以分为组合电路和时序电路两种。

组合逻辑电路是指输出值仅由输入信号的状态决定的电路。组合逻辑电路的输出不依赖于过去的输入。也就是说,不需要记忆维持过去的输入信号,因此不含有存储元件。

时序电路是指输出值同时依赖于现在和过去输入信号的逻辑电路。时序电路中含有用于保持输入的存储元件。

1.3.9 时钟同步设计

时钟同步设计是一种数字电路的设计技术。前文提到过,时序电路的输出同时取决于现在和过去的输入。但如何区别现在和过去呢?

在时钟同步设计中,有一种周期性地在 H 和 L 间变化的时钟信号,时钟变化边沿(上升沿或下降沿)之前被称为过去,之后被称为现在。时钟同步设计中,由时钟边沿触发同步更新电路的状态。时钟同步设计最大的优点是,设计者只需要注意时钟边沿的时序,电路的设计和验证都比较容易。因此很多数字电路都是时钟同步设计。

1.3.10 小结

本节介绍了数字电路的基础。在数字电路中使用 1 和 0 表现信息,基于用 MOSFET 组合构成的 CMOS 来实现各种逻辑电路。近年来,绝大多数的计算机都是基于数字电子电路实现的。

专栏

数字电路相关书籍

論理回路の設計(浅川毅著、コロナ社)(中文译名《逻辑 电路的设计》)

这本书详细讲解了逻辑电路的原理和设计方法。主要面向学习 逻辑电路设计的学生和技术员,也可作为大学或大专院校信息 专业学生的教材,非常适合初学者。

• ディジタル設計者のための電子回路(天野英晴著、コロナ社)

(中文译名《面向数电设计者的电路》)

这本书讲解了数字电路中的电路相关知识和设计技术。与《逻辑电路的设计》一书相比,本书对电路和电磁方面知识的讲解 更通俗易懂。本书也可以作为大学和大专院校信息专业学生的 教材。

1.4 Verilog HDL 语言

本节将讲述 Verilog HDL 语言的基础知识,也会一并介绍基于 Icarus Verilog 和 GTKWave 的仿真环境。本书使用的 Verilog HDL 是基于 Verilog HDL 2001 标准的语言 规范。这一节主要说明 Verilog HDL 的基础语法,读者们可以跳跃阅读,在读写代码需要的时候再翻回来查阅。

1.4.1 什么是 Verilog HDL

Verilog HDL 是一种 HDL 语言(Hardware Description Language,硬件描述性语言)。使用 Verilog HDL 语言可以进行抽象度较高的 RTL(Register Transfer Level,寄存器传输级)电路设计。RTL 是根据寄存器间的信号流动和电路逻辑来记述电路动作的一种设计模型。

很早以前,电路设计是将一个个逻辑与、逻辑或等门电路绘制在电路图纸上。但随着半导体技术的发展,这种方式很难高效地实现大规模硬件的设计。如今的电路设计通常采用 RTL 模型。

图 1-33 是一个使用 Verilog HDL 进行硬件设计的流程示例。首先,在硬件功能确定之后,使用 Verilog HDL 语言进行目标电路和测试程序的编写。同时根据硬件的设计目标设定面积、时钟周期等约束参数。然后在仿真器上使用测试程序对设计好的电路进行功能验证。最后,验证成功的 Verilog HDL 在约束参数条件下进行逻辑综合并生成电路网表。

逻辑综合是将RTL级别记述的抽象电路转换到门电路级别的电路网表的过程。逻辑综合时,针对 ASIC(Application Specific Integrated Circuit)、FPGA(Field Programmable Gate Array)等不同电路实现技术,需要使用这些技术厂商提供的相应的目标元件库。

图 1-33 展示的是一条自上而下的单向设计流程,当发生电路验证失败、逻辑综合结果无法满足约束条件(无法收敛)等情况时,需要更正设计或参数并返回到设计的上流重新开始。电路网表生成以后还有布局布线等过程,在此不作阐述。

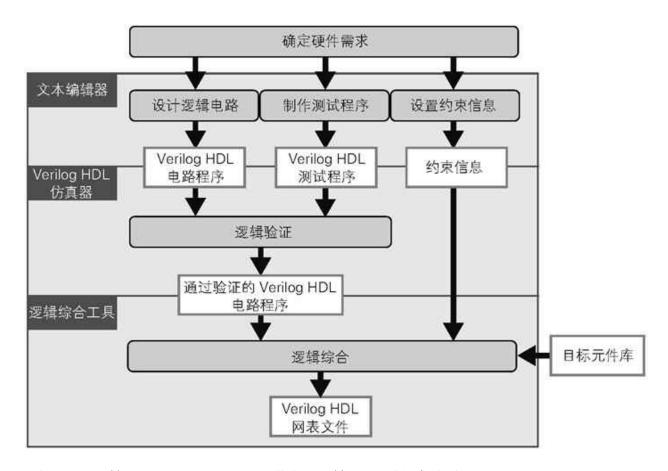


图 1-33 使用 Verilog HDL 进行硬件设计的流程例

1.4.2 电路描述

本节讲述如何使用 Verilog HDL 进行电路的描述。

• 模块

Verilog HDL 中使用模块来设计一个功能单位的逻辑。模块也是 Verilog HDL 语言中最基本的构成单位。模块声明的语法如图 1-34 所示。

endmodule

图 1-34 模块声明的语法

下面,我们一起来看一下如何使用模块来描述一个 32 位的加法器。将要实现的加法器有 in_0 和 in_1 两个 32 位的输入信号,它们相加的结果从 32 位的 out 信号输出。图 1-35 是该加法器的框图,图 1-36 展示了它的程序代码。

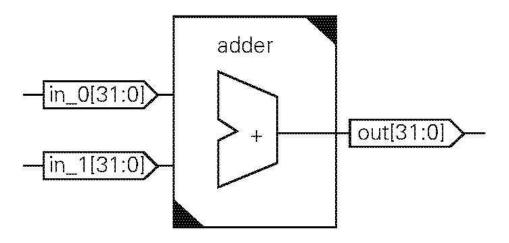


图 1-35 加法器的框图

```
module adder (
    input wire [31:0] in_0, // 输入0
    input wire [31:0] in_1, // 输入1
    output wire [31:0] out // 输出
);
    // in_0 和in_1 相加后结果代入out
    assign out = in_0 + in_1;
endmodule
```

图 1-36 加法器的程序

模块声明的语法是在 module 关键字后记述该模块名。图 1-36 中的示例使用 adder 作为模块名。在紧随模块名的圆括号中对该模块的输入输出信号进行定义。输入信号的声明使用 input 关键字,输出信号的声明使用 output 关键字,双向信号的声明使用 inout 关键字进行描述。信号声明的关键字后分别要对数据类型、信号线的位宽和信号名进行描述。变量位宽的定义是在方括号中记述最高位和最低位的位置,中间用冒号隔开,如 [31:0]。比特数据的最高位被称为 MSB(Most Significant Bit),最低位称为 LSB(Least Significant Bit)。图 1-36 中声明的 in_0 和 in_1 信号是 32 位 wire型输入信号,out 是 32 位 wire型输出信号。输入输出信号的声明之后使用右圆括号加分号结束,如);。接下来对模块内的电路逻辑进行描述。在图 1-36 的示例中,使用 assign 语句,将 in_0 和 in_1 相加的结果输出到 out。电路逻辑记述完毕,最后使用 endmodule关键字结束模块的定义。

Verilog HDL 是自由格式语言,可以在任意地方加入换行、空格以及 Tab 等空白符号。另外,因为 Verilog HDL 语言区分大小写,所以大小写的英文字符分别表示不同的含义。有效的标识符包括英文字母(a~z, A~Z)、数字(0~9)、下划线(_)和美元符号(\$)。标识符可以用来命名变量和模块。用户自定义的标识符必须以英文字母或下划线开头。Verilog HDL 语言中,在 /* 和 */ 之间,或从 // 开始到一行末尾的文字被视为注释。 begin 和 end 之间的部分称为块。

• 模块的实例化

设计好的模块可以被其他模块调用。模块实例化的方法如图 1-37 所示。该示例调用了图 1-36 中实现的加法器。使用分层的设计方式可以将复杂的电路分割成多个功能单元简化设计,也有助于增强代码的可维护性和移植性。

```
adder adder01 (
    .in_0 (adder01_in_0), // adder01_in_0 信号连接到in_0 端口
    .in_1 (adder01_in_1), // adder01_in_1 信号连接到in_1 端口
    .out (adder01_out) // adder01_out 信号连接到out 端口
);
```

图 1-37 模块的实例化

• 逻辑值与常数表达

Verilog HDL 中可以使用的逻辑值如表 1-4 所示。逻辑值可以表达为 0 和 1。当由于复位等操作后未经初始化或因设计问题无法确定是 0 还是 1 时,使用不定值 x 来表达。此外,电气概念上的绝缘状态(没有任何连接)被称为高阻状态,用 z 来表示。

表 1-4 Verilog HDL 的逻辑值

逻辑值	名称	含义
0	Low	数值0(接地)与逻辑假
1	High	数值1(电源电压)与逻辑真
X	不定值	无法确定值是0还是1
Z	高阻值	电气绝缘状态

常数的格式如图 1-38 所示。首先在位宽中指定常数的宽度,然后是单引号加表示该常数为几进制的底数符号。二进制底数符号为b、八进制为 o、十六进制为 h。最后在数值中指定该常数的数值。图 1-38 中的示例说明了十进制数 60 的各种表达方式。

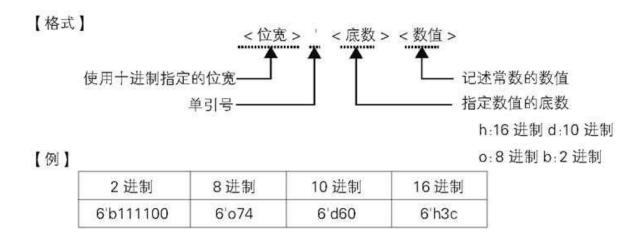


图 1-38 常数的格式与示例

• 变量的声明与数据类型

变量声明的格式如图 1-39 所示。数据类型和变量名是必要项目, 其他项可以省略。符号和位宽如果省略则根据数据类型设置为默认 值。元素数省略默认声明元素数为 1 的变量。

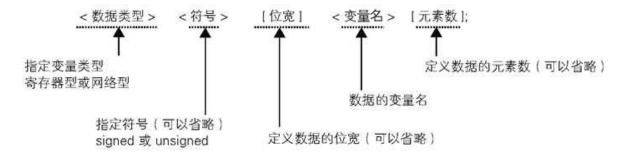


图 1-39 变量声明的格式

数据类型有寄存器型和网络型两种。寄存器型是可以保存上次写入数据的数据类型,根据程序不同可以生成锁存器、触发器等存储元件,也可能生成组合电路。寄存器型变量如表 1-5 所示。本章主要使用 reg 和 integer 两种类型。

表 1-5 寄存器型变量

名称	默认位宽	默认符号	含义
reg	1位	无符号	比特数据
integer	32 位	有符号	整数

寄存器型变量可以在接下来将要介绍的 always 和 initial 语句中实现过程赋值(Procedural Assignment)。这种方式称为过程赋值。过程赋值分为阻塞式和非阻塞式赋值两种。

阻塞式赋值是一种按照代码顺序进行赋值的方式。在先赋值的代码 赋值完成之前阻塞后续代码的赋值,因此得名阻塞式赋值。阻塞式 赋值使用 = 运算符。

非阻塞式赋值中所有代码不会互相阻塞,同时进行赋值。非阻塞式赋值使用 <= 运 算符。

在一个过程块中,阻塞式赋值和非阻塞式赋值只能使用其中一种。 阻塞式赋值的格式如图 1-40 所示。非阻塞式赋值的格式如图 1-41 所示。

【格式】

<左值 > = < 表达式 >:

【例】

a = a + 1; 由上到下计算 b = a + 1;

赋值前 a 的值是 0 的话, 赋值后 a 为 1, b 为 2。

图 1-40 阻塞式赋值

【格式】

<左值 > <= < 表达式 >;

【例】

赋值前 a 的值是 0 的话, 赋值后 a 和 b 都是 1。

图 1-41 非阻塞式赋值

网络型是用来描述模块和寄存器间连接的数据类型。网络型只描述信号的传输不持有数据。表 1-6 对网络型变量进行了说明。本章只使用 wire 型。

表 1-6 网络型变量

名称	默认位宽	默认符号	含义
wire, tri	1位	无符号	线连接
wor, trior	1位	无符号	线或连接
wand, triand	1位	无符号	线与连接
tri1, tri0	1位	无符号	有上拉或下拉的连接
supply0, supply1	1位	无符号	接地或接电源的连接

网络型变量可以在 assign 语句或声明语句中实现连续赋值

(Continuous Assignment)。连续赋值就是进行连续的赋值。图 1-42 给出了 assign 语句中连续赋值的格式和示例。图 1-43 给出了声明语句中连续赋值的格式和示例。

【格式】

assign < 网络型变量> = < 表达式>;

【何】

wire [31:0] word;

```
wire [7:0] byte0, byte1, byte2, byte3;
assign byte0 = word[31:24];
assign byte1 = word[23:16];
assign byte2 = word[15:8];
assign byte3 = word[7:0];
```

图 1-42 assign 语句中连续赋值

```
【格式】
< 网络类型> (符号) (位宽) < 变量名> = < 表达式>;

【例】
wire [31:0] word;
wire [7:0] byte0 = word[31:24];
wire [7:0] byte1 = word[23:16];
wire [7:0] byte2 = word[15:8];
wire [7:0] byte3 = word[7:0];
```

图 1-43 声明语句中连续赋值

变量的符号用 signed 和 unsigned 关键字指定。在赋值或比较等处理时,如果需要在有符号数和无符号数间进行转换,需要使用 \$signed() 和 \$unsigned() 系统任务(system task)。无符号数转换为有符号数时使用 \$signed(),有符号数转换为无符号数时使用 \$unsigned()。变量声明的示例如图 1-44 所示。

图 1-44 变量声明示例

专栏

默认网络类型

使用网络型变量时,如果定义默认网络类型,可以不用声明直接使用。引入这种方式,是为了可以在大量使用网络型变量的网表程序中减少代码量。但是默认网络类型也有副作用。由于失误而未声明类型的信号会被自动处理为默认网络类型,编译器无法检测错误。

默认网络类型由编译器指示词 `default_nettype 指定。图 1-45 给出了默认网络类型的格式与示例。默认网络类型为 none 时,不启用默认网络类型。RTL 设计时为了规避默认网络类型的副作用,推荐将默认网络类型设置为无效。

图 1-45 默认网络类型的指定

• 运算符

Verilog HDL 中的运算符如表 1-7 所示,运算符的优先级如图 1-46 所示。运算首先根据运算符优先级的高低顺序执行,优先级相同的运算符按照从左到右的顺序执行。使用圆括号可以改变运算的优先顺序。

表 1-7 Verilog HDL 中的运算符

种类	运算符	含义	优先级
	+	加法	3 (符号1)
算术运算符	-	减法	3 (符号1)
	*	乘法	2
	/	除法	2
	%	求余	2
	~	NOT	1
	&	AND	7
位运算符		OR	8
	٨	XOR	7
	~^	XNOR	7
	&	AND	1
	~&	NAND	1
烷减 异管效		OR	1
缩减运算符	~	NOR	1
	٨	XOR	1
	~^	XNOR	1
移位运算符	<<	逻辑左移	4
	>>	逻辑右移	4
	==	相等	6
等式运算符	!=	不等	6
守八씯昇刊	===	相等(x,z 也参与比较)	6
	!==	不等(x,z 也参与比较)	6
	>	大于	5
关系运算符	<	小于	5
大尔 巴 异的	>=	大于等于	5
	<=	小于等于	5
	!	逻辑非	1
逻辑运算符		逻辑或	9
	&&	逻辑与	10
三项运算符	?:	条件运算	11
拼接运算符	{}	拼接	-

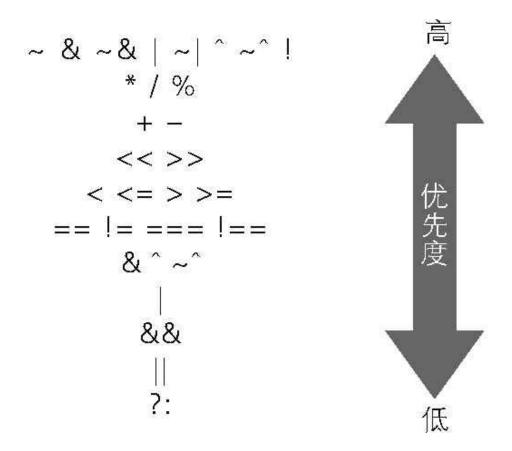


图 1-46 运算符的优先顺序

缩减运算符的特点是对信号的所有位进行位运算,最终输出1位的运算结果。缩减运算符的说明如图 1-47 所示。

```
wire [3:0] a;
wire b;
assign b = &a; // 与a[3] & a[2] & a[1] & a[0] 等价
```

图 1-47 缩减运算符示例

```
【格式】 { 比特序列0, 比特序列1, ..., 比特序列N } 【例】 wire [7:0] byte0, byte1, byte2, byte3;
```

```
wire [31:0] word = {byte0, byte1, byte2, byte3};
将word[31:24] 赋值给byte0, 将word[23:16] 赋值给byte1, 将word[15:8]
赋值给byte2, 将word[7:0] 赋值给byte3
```

图 1-48 使用拼接运算符组合比特序列

```
【格式】
{ 重复次数{ 被重复数据}}
【例】
wire [7:0] byte;
wire [31:0] word = {4{byte}};
word[31:24]、word[23:16]、word[15:8]、word[7:0]
全都赋予byte 变量的值
```

图 1-49 使用拼接运算符重复比特序列

• 条件分支语句 if 与 case

条件分支可以使用 if 或 case 语句实现。if 和 case 语句的语法格式与示例分别如图 1-50 和图 1-51 所示。

图 1-50 if 语句的格式与示例

```
【格式】
case (< 表达式>)
   < 表达式> : < 语句序列> < 表达式>, < 表达式>, ... : < 语句序列>
                                : 〈语句序列〉
   default
endcase
【例】
case (data[3:0])
   4'h0
                                : begin
       ... // data[3:0] 为4'h0 时的语句
   end
   4'h1, 4'h2
       ... // data[3:0] 为4'h1 或4'h2 时的语句
   default
                                : begin
     ... // 默认语句
   end
endcase
```

图 1-51 case 语句的格式与示例

if 语句括号中的条件成立时,执行其中的语句序列。当含有 else 语句,if 条件不成立时,执行 else 中的语句。else 语句中也可以再使用 if 进一步限制条件。case 语句是执行与括号中条件相等的表达式内的语句序列。if 和 case 语句可以在 initial 或 always 语句声明的过程块中使用。

• 循环语句 for 与 while

使用 for 和 while 语句可以实现循环操作。for 和 while 语句的语法格式与示例分别 如图 1-52 和图 1-53 所示。

```
【格式】
for (< 赋值语句>; < 表达式>; < 赋值语句>) < 语句序列>
【例】
for (i = 0; i < 10; i = i + 1) begin
    ... // 重复执行10 次
end
```

图 1-52 for 语句的格式与示例

图 1-53 while 语句的格式与示例

for 语句在圆括号中央的表达式条件成立时执行其中的语句序列。 第一次进入 for 语句时,执行圆括号内左边语句并进入重复过程。 第二次循环开始先执行圆括号内中央的表达式,如果为真则先执行 循环体内的语句序列,随后执行圆括号内右边的语句。然后再次执 行圆括号内中央的表达式,为真的话继续重复上述操作。while 语 句是在圆括号中表达式为真时重复执行其中的语句序列。for 和 while 语句可以在 initial 或 always 语句声明的过程块中使用。

• always 过程块

always 是为了描述过程块而存在的语句。always 的语法格式如图 1-54 所示。

always @(< 事件表达式>) < 语句序列>

```
always #< 常数表达式> < 语句序列>
```

图 1-54 always 语句的格式

当指定 always 语句中的事件表达式时,所指定的事件触发时执行其中的语句序列。事件可以是特定信号的变化、信号的上升沿(posedge)、信号的下降沿(negedge)等。always 语句中如果使用常数,则会在每经过该常数时间便执行一次 always 中的语句序列。这个功能主要是在仿真时使用。always 过程中可以使用寄存器变量赋值、if、case、for、while 等语句。

• 使用 always 语句描述组合电路

使用 always 语句描述组合电路时,事件表达式描述方式如图 1-55 所示。事件表达式中写入通配符 *。这样一来,任何输入信号变化时都会执行过程块中的代码。

图 1-55 使用 always 语句描述组合电路

示例中定义了一个 adder 模块,它有两个 32 位 wire 型输入 in_0 和 in_1、一个 32 位 reg 型输出。always 中将两个输入相加后赋值给了输出。这里使用了阻塞式赋值。

专栏

组合电路描述中锁存器的推定与 Don't care

使用 always 语句描述组合电路时,如果信号未被赋值,有可能会引入锁存器。以图 1-56 所示的代码为例,当 in 为 2'b11 时会发生什么情况呢?这时由于没有赋值,out 的值不发生变化,也就是说会保持之前的值。而为了保持之前的值就需要存储元件。这时会生成异步的存储元件锁存器。因此虽然设计的是组合电路,但产生了本不应有的存储元件,变成了时序电路。

```
module bin_decoder (
    input wire [1:0] in,
    output reg [3:0] out
);

always @(*) begin
    case (in)
        2'b00 : out = 4'b0001;
        2'b01 : out = 4'b0010;
        2'b10 : out = 4'b0100;
        // 没有2'b11 情况的描述
    endcase
    end

endmodule
```

图 1-56 寄存器推定示例

寄存器推定的发生原因是不完整的 case 语句或没有 else 的 if 语句。为了规避这个问题,一定要将 case 语句的条件写全,

或者使用 default 来确定默认值。并且,使用 if 语句时一定要写 else 条件,或者在 if 语句前为变量值赋予默认值。

也存在这种情况:确定不存在 case 和 if 语句设定之外的条件,或者设定条件之外随便输出什么都可以。这种情况称为 Don't care(忽略),输出为逻辑综合时优化的数值。Verilog HDL 中 Don't care 的指示方法是在 default 中为输出赋予不定值。图 1-56 的示例中加入 Don't care 指示的程序如图 1-57 所示。

```
module bin_decoder (
    input wire [1:0] in,
    output reg [3:0] out
);

always @(*) begin
    case (in)
        2'b00 : out = 4'b0001;
        2'b01 : out = 4'b0010;
        2'b10 : out = 4'b0100;
        default : out = 4'bxxxx;
    endcase
    end
endmodule
```

图 1-57 Don't care 指示方法

• 使用 always 描述时序电路

使用 always 语句描述时序电路时,事件表达式描述方式如图 1-58 所示。时序电路含有触发器等存储元件,基本上都是按照时钟同步执行。因此事件表达式中要指定时钟的信号边沿和时钟信号名。

```
【格式】
always @(< 边沿> < 信号> [or …]) begin
    … // 记述时序电路
end

【例】
module ff (
    input wire clk, // 时钟
```

```
input wire reset_, // 复位(负逻辑)
input wire d_in, // 输入的数据
output reg d_out // 输出的数据
);

always @(posedge clk or negedge reset_) begin
    if (reset_ == 1'b0) begin // 异步复位
        d_out <= 1'b0;
    end else begin // 数据的储存
        d_out <= d_in;
    end
end

end

end
```

图 1-58 使用 always 语句记述时序电路

时钟信号边沿是指确定在时钟信号上升时触发电路动作,或者在时钟信号下降时触发电路动作。上升时动作记述为 posedge,下降时动作记述为 negedge,然后记述信号名。事件表达式还可以使用 or 列举多个条件。为存储元件设置异步复位(reset)信号时,除了时钟信号还要写上复位信号的边沿和信号名。

图 1-58 的示例中定义了一个叫做 ff 的模块,它有 clk、reset_、d_in 三个一位 wire 型输入信号,和一位 reg 型输出信号 d_out。d_out 在 clk 的上升沿同步动作,将 d_in 的值储存。并且 d_out 在 reset_ 的 下降沿被异步地复位,初始化为 0。

• 预处理

预处理是在代码编译前对其进行预先处理的程序。Verilog HDL 中的预处理可以实现宏定义和条件编译。预处理使用编译指示符可对编译器进行各种控制。图 1-59 介绍了本书用到的编译指示符的格式和示例。

图 1-59 编译指示符

编译指示符以后引号(`)开头。使用 `include 语句可以插入引用文件。使用 `define 语句可以进行宏的定义。在图 1-59 的示例中,定义了名为 BYTE_DATA_W、值为 8 的宏。Verilog HDL 中为了区分宏与变量名,宏的名称前也加有后引号(`)。代码中使用宏时,要像 `BYTE_DATA_W 一样记述。

使用 `ifdef 和 `ifndef 可以实现条件编译。`ifdef 是在指定的宏存在的条件下,执行 `ifdef 到 `endif 的代码。`ifndef 是在指定的宏不存在的条件下,执行 `ifndef 到 `endif 的代码。两者都可以使用 `else 指定不满足条件时执行的动作。

图 1-59 的示例中,宏 TEST1 存在时执行从 `ifdef 到 `else 的代码,不存在时执行从 `else 到 `endif 的代码。宏 TEST2 不存在时执行从 `ifndef 到 `else 的代码,存在时执行从 `else 到 `endif 的代码。

• 程序例

下面利用前面介绍的 Verilog HDL 语法,演示一个代码示例。本示例制作了 32 组位宽为 32 的寄存器堆。图 1-60 是寄存器堆的框图。寄存器堆中有作为存储的 32 个 32 位寄存器,以及读写寄存器序列用的接口。

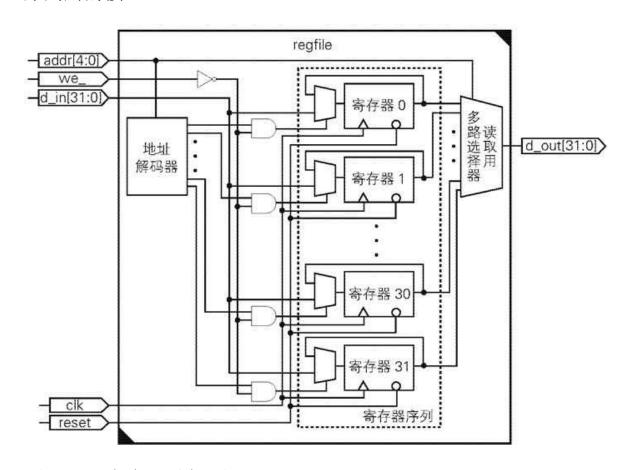


图 1-60 寄存器堆框图

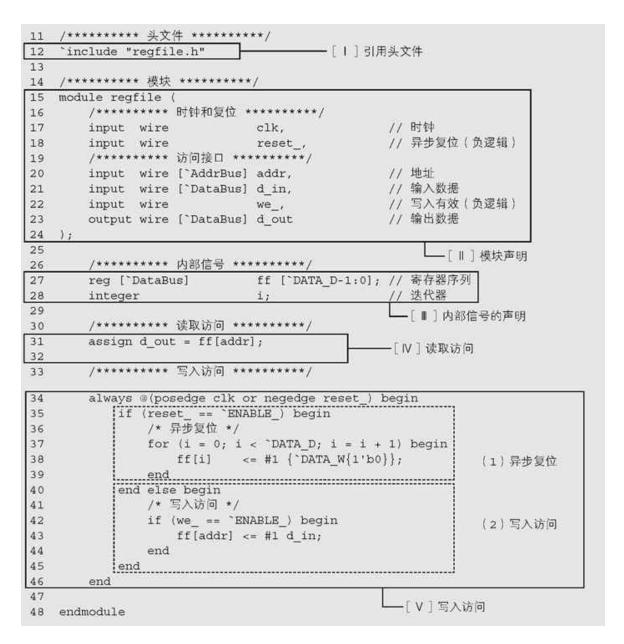
读取操作时将地址信号(addr)指定的寄存器的内容,通过多路选择器选择,输出到输出信号(d_out)。对于写入操作,当写入使能信号(we_)有效时,向地址信号(addr)指定的寄存器写入输入数据(d in)。

寄存器堆模块在 regfile.v 文件中实现。regfile.v 引用了 regfile.h 头文件。regfile.h 的内容在代码 1-1 中列出,regfile.v 的内容在代码 1-2 中列出。

代码 1-1 头文件示例 (regfile.h)

```
11
  `ifndef REGFILE HEADER
                                      // 包含文件防范
      `define __REGFILE_HEADER__
12
13
      /***** 信号电平 *******/
14
15
      `define HIGH
                              1'b1
                                      // 高电平
16
      `define LOW
                              1'b0
                                     // 低电平
17
     /******* 逻辑值 ******/
18
19
      `define ENABLE
                             1'b0
                                      // 有效(负逻辑)
                             1'b1
20
                                      // 无效(负逻辑)
      `define DISABLE
21
      /****** 数据 ******/
22
23
                                      // 数据宽度
      `define DATA W
                              32
                                      // 数据总线
      `define DataBus
24
                              31:0
25
                                      // 数据深度
      `define DATA D
                              32
26
      /******* 地址 *******/
27
28
      `define ADDR W
                              5
                                      // 地址宽度
      `define AddrBus
                                      // 地址总线
29
                             4:0
30
31
  `endif
```

代码 1-2 代码示例(regfile.v)



[I] 引用头文件

引用头文件的语句写在模块之外。

[II] 模块声明

声明 regfile 模块并定义输入输出接口。

[III] 内部信号的声明

定义寄存器序列 ff 和 for 语句的 i(循环的计数器)。

「IV〕读取访问

将 addr 指定地址的寄存器序列的值连续赋值给 d out。

「V]写入访问

- 1. 中进行异步复位操作。reset 信号使能时,使用 for 循环将全部 ff 的值初 始化为 0。
- 2. 中进行写入访问操作。we 信号使能时,将输入信号 d in 的值写 入addr地址指定的寄存器序列中。

专栏

正逻辑与负逻辑

控制信号的有效、无效与信号高低电平相对应时,高电平有 效、低电平无效的分配方式称为正逻辑。反之,高电平无效、 低电平有效的分配称为负逻辑。

不论信号电平的高低,控制信号转为有效状态的动作称为 assert (断言),转为无效状态的动作称为 negate (无效)。并 目,信号有效时称为 enable (使能),信号无效时称为 disable (非使能)。

1.4.3 电路仿真

使用 Verilog HDL 不仅可以设计电路,还可以对所设计的电路进行 仿真。通过仿真可以实现逻辑验证,从而测试设计好的电路是否可 以正确工作。记述仿真程序的文件称为 Testbench。下面,我们来 看一下 Testbench 的制作方法。

○ Testbench 的构造

Testbench 是对制作的电路进行仿真、测试的模块。Testbench 的构造如图 1-61 所示。

图 1-61 Testbench 的构造

Verilog HDL 中的 Testbench 本身就被定义为一个模块。通常,Testbench 没有输入输出信号。Testbench 调用被测模块,传递输入信号并观测输出。被测模块输入输出端口上的信号作为 Testbench 的内部信号进行定义。通常,输入端口为了将值带入使用寄存器型变量,而输出信号接网络型变量对输出值进行观测。Testbench 中,使用 initial 语句生成测试用例,然后观测模块的输出。

在 Testbench 中,`timescale 用来设定仿真执行的时间单位。 `timescale 的设定中使用数字和单位(fs、ps、ns、us、ms、 s) 指定单位时间和时间精度。图 1-62 列出了 `timescale 的使 用格式。

```
`timescale < 单位时间>/< 时间精度>
```

图 1-62 Testbench 的格式

单位时间用来指定仿真的一个单位时间相当于多少秒。时间精度用来表示仿真处理的时间精度,并根据时间精度取数值的近似值。没有必要取过小的时间精度,这会延长仿真时间。单位时间和时间精度的关系必须满足"单位时间≥时间精度"。

。 用 initial 语句生成测试用例

initial 语句是在仿真开始时只会执行一次的语句。initial 语句的格式与示例如图 1-63 所示。initial 语句和延迟描述组合,可以用来生成测试用例。

```
【格式】
initial begin
                    // 过程的描述
end
【例】
initial begin
                    // 时刻0 时执行
   #0 begin
   end
                    // 时刻10 时执行
   #10 begin
   end
                    // 时刻20 时执行
   #10 begin
   end
end
```

图 1-63 initial 的格式与示例

。 延迟语句

#字符用来记述延迟语句,其格式与示例如图 1-64 所示。延迟语句中指定的数值意味着 `timescale 中设定单位时间的个数。延迟语句只用在仿真程序中,用来在特定时间延迟后施加信号

并生成测试用例。不会对逻辑综合的结果产生影响。

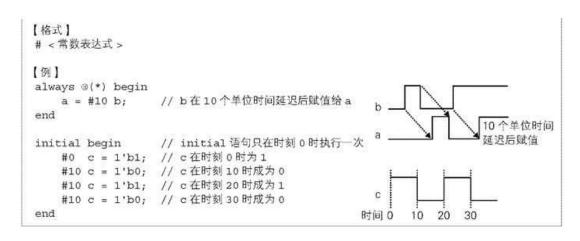


图 1-64 延迟语句的格式与示例

专栏

同步电路中信号变化的时序

仿真与时钟同步的电路时需要注意信号变化的时序。

图 1-65 所示的电路,时刻 10 的时候 out 信号值会变成什么呢?如果是在信号变化之前值为 L,如果是在信号变化之后则值为 H。实际上,结果是哪个值取决于仿真器。由于仿真中将信号变化的延迟作为 0 来处理,因此会引起这样的问题。在实际电路中,从时钟信号上升到信号变化之间会产生一定时间的延迟。因此时刻 10 时的值应该是 L。

```
module osc (
    input wire clk,
    input wire reset_,
    output reg out
                                                          out 的值是 0 还是 1?
);
    always @(posedge clk or negedge reset ) begin
        if (reset_ == 1'b0) begin
                                                        clk
            out <= 1'b0;
        end else begin
            out <= ~out;
                                                     reset
        end
    end
                                                       out
                                                      时间0
end
                                                                  10
```

图 1-65 同步电路信号时序示例

仿真时为了避免这种情况,如图 1-66 所示,通常使用延迟语句将信号变化的时序向后顺延一个时间单位。由此,时刻 10 时 out 的值依然是 L。这种记述方式不会对逻辑综合的结果产生影响。

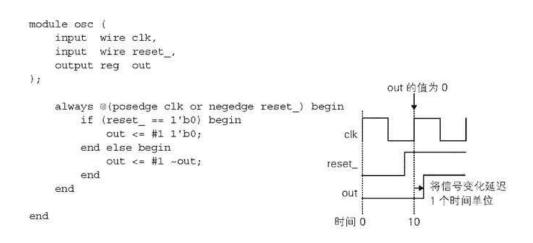


图 1-66 信号变化时序的延迟

。 时钟的生成

如果被测模块要用到时钟,需要在 Testbench 中生成时钟信号。图 1-67 展示了一种使用 always 语句生成时钟的方法。always 语句中指定常数作为时间间隔,每经过这个时间间隔就会执行一次 always 中的语句。也就是说,图 1-67 中的 clk 信号值每过 10 个单位时间就会翻转。用这个方式就可以生成时钟信号。需要注意的是,clk 应该在 initial 语句中的时刻 0 时被初始化。

图 1-67 时钟的生成

。 系统任务

通过使用 Verilog HDL 预置的系统任务,可以达到控制仿真、输出字符串等目的。下面列出了一些经常用到的系统任务。

■ \$display("含有格式的字符串",...)

根据第1参数中含有格式的字符串,将第2参数开始的任意个数的参数格式化,最后加换行符输出。

■ \$write(" 含有格式的字符串 ",...)

与 \$display 功能相同,但输出后不换行。

\$time

返回目前的仿真时间。

\$finish

结束仿真。

。 载入存储镜像

仿真时,有时需要向存储器等读入预先准备好的数据。可以使用 \$readmemh 系统任务从文件中读入数据并设置存储器。 \$readmemh 格式如下所示。

■ **\$readmemh(**"文件名 ", 读入对象)

第 1 参数所指定文件的数据读入第 2 参数指定的存储器中。 \$readmemh 使用的存储镜像文件使用十六进制文本文件记录。每一行记录一个地址的数据。图 1-68 是读入

存储镜像的示例。

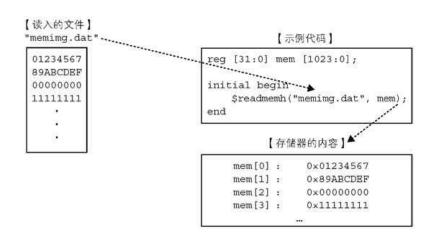


图 1-68 存储镜像读入示例

。波形的输出

仿真时的信号变化可以输出到波形文件中。波形文件有很多种,本书将介绍多数波形软件都支持的 VCD 格式波形文件的输出方法。

VCD 文件的输出使用 \$dumpfile 和 \$dumpvars 两个系统任务来实现。波形输出的格式和示例如图 1-69 所示。在 initial 中调用 \$dumpfile 和 \$dumpvars,可以实现波形文件的输出。

```
【格式】
$dumpfile(< 文件名>)
$dumpvars(< 开始时刻>, < 输出波形的模块名或信号名>)

【例】
initial begin
$dumpfile("test.vcd"); // 将波形输出到test.vcd 文件
$dumpvars(0, test); // 从时刻0 开始输出模块test 的波形
end
```

图 1-69 波形文件的格式与示例

。 **Testbench** 实例

接下来,我们为程序示例中实现的寄存器堆制作 Testbench。 代码 1-3 中列出了 Testbench 的代码。

代码 1-3 Testbench 示例(regfile_test.v)

```
/******** Time scale *******/
                                        // Time scale
12
    `timescale lns/lps
                                        [ | ] 定义 Time scale
13
    /*******************/
`include "regfile.h"
                                            -----[ II ] 引用头文件
15
16
    /******** 模块 *******/
20
        // 时钟&复位
21
                       clk;
        reg
                                        // 时钟
22
                                       // 复位(负逻辑)
       reg
                      reset ;
       // 访问接口
23
        reg ['AddrBus] addr;
reg ['DataBus] d_in;
24
                                        // 地址
                                        // 输入数据
// 写入使能(负逻辑)
25
26
        reg
                       we_;
                                        // 输出数据
        wire [ DataBus] d_out;
27
                                                           [ IV ] 定义内部信号
28
        /********* 内部变量 *******/
29
        integer
                                        // 迭代器
                        i;
30
        /****** 定义仿真循环 ********/
        parameter STEP = 100.0000; // 10 M
31
        /****** 生成时钟 *******/
33
34
      always #(STEP / 2) begin
35
            clk <= ~clk;
                                         _____[ V ] 生成时钟
36
        /********* 实例化测试模块 *******/
38
        regfile regfile (
/************ 时钟&复位 *********/
39
40
41
            .clk (clk),
reset (reset_),
                                       // 时钟
42
            .reset_ (reset_), //
/********** 访问接口 ********/
                                         // 复位(负逻辑)
43
            .addr (addr),
.d_in (d_in),
.we_ (we_),
.d_out (d_out)
                                      // 地址
44
45
                                        // 输入数据
                                                           [VI]实例化测试模块
                                       // 写入使能(负逻辑)
// 输出数据
46
47
48
49
                                                                    - [ VII ] 测试用例
        /******* 測试用例 *******/
50
51
        initial begin
           52
53
54
55
56
            we_ <= `DISABLE_;
end</pre>
57
            # (STEP * 3 / 4)
59
            # STEP begin
reset_ <= `DISABLE_, (2)解除复位 (3)读写验证end
60
61
62
            # STEP begin
| for (i = 0; i < `DATA_D; i = i + 1) begin
63
64
65
                    # STEP begin
                       addr <= i;
d_in <= i;
66
67
                        we_ <= `ENABLE_;
68
69
                   end
70
                    # STEP begin
                       TEP begin
  addr <= { `ADDR_W{1'b0}};
  d_in <= { `DATA_W{1'b0}};
  we_ <= `DISABLE_;
  if (d_out == i) begin</pre>
71
72
73
74
75
                           $\text{Sdisplay}(\$\text{time}, " ff[\$d] Read/\text{Write Check OK !", i);
76
                        end else begin
77
                           $display($time, " ff[%d] Read/Write Check NG !", i);
                       end
78
79
80
81
             end
82
            # STEP begin
               Sfinish; (4)结束仿真
83
84
        end
85
86
         /********** 输出波形 *******/
87
        initial begin
88
            $dumpfile("regfile.vcd");
89
                                                                -[VII]输出波形
            $dumpvars(0, regfile);
90
91
        end
92
93 endmodule
```

[I] 定义 Time scale

单位时间为 1ns, 时间精度为 1ps。

[II] 引用头文件

引用 regfile.h。

[III] 声明模块

声明 regfile_test 模块。Testbench 没有输入输出端口。

[IV] 定义内部信号

被测模块的输入端口连接 reg 型变量,输出端口连接 wire 型变量。为仿真循环定义了名为 STEP 的参数,STEP 的值为 100ns。

[V] 生成时钟

这里生成了频率为 10MHz 的时钟。10MHz 的时钟每 50ns(STEP/2)在 H 与 L 间重复一次。

[VI] 实例化测试模块

实例化 regfile。

「VII]测试用例

(1)处的时刻 0 时对信号线进行初始化。(2)处的语句解除复位信号。(3)处对寄存器堆的读写进行验证。最初一次 STEP 时,对寄存器地址 i 处写入数值 i,下一个 STEP 时将其读出,并用 if 语句比较、验证读出的结果是否正确。 这个过程使用 for 语句重复 32 次。(4)处结束本次仿真。

[VII] 输出波形

在 initial 中输出仿真波形。将实例化的 regfile,从时刻 0 开始

的波形输出到 regfile.vcd 文件中。

1.4.4 Verilog HDL 的仿真环境

本节对 Verilog HDL 语言的仿真环境进行说明。Verilog HDL 仿真器工具有很多,本书仅介绍 Icarus Verilog。本书还会介绍可以查看仿真生成的波形文件的工具 GTKWave。这两个工具都是自由软件,并且可以在 Windows、Linux 等各种平台运行。

■ 下载与安装

Icarus Verilog 与 GTKWave 官方网站 URL 如下所示:

Icarus Verilog

http://iverilog.icarus.com/

GTKWave

http://gtkwave.sourceforge.net/

从这些网站可以下载并安装上述两个软件。Icarus Verilog for Windows 页面提供了这两个软件捆绑的安装程序,本书使用这种安装方式。

Icarus Verilog for Windows

http://bleyer.org/icarus/

访问上述 URL 就会打开如图 1-70 所示的页面。从 Download 下方列出的链接下载最新版的安装程序。图 1-70 中 iverilog-0.9.5_setup.exe [6.84MB] 的链接为最新版。 接着运行下载的安装文件,并根据安装向导安装程序。不 需要指定特殊的参数,默认安装即可。



图 1-70 Icarus Verilog for Windows

Icarus Verilog 仿真功能需要从命令行执行。为了确认 Icarus Verilog 已经正确安装,我们打开命令行窗口执行一 下。

要打开命令行窗口,先按 windows 键,依次点击所有程序 → 附件 → 命令提示符。之后会打开图 1-71 所示的黑色画面。画面中显示的文字"C:\Users\Kazutoshi Suito>"称为提示符,是提示用户输入命令的信息。提示符的前半部分提示的是当前目录。Windows 中的目录称为文件夹。用户可以在提示符后输入命令。

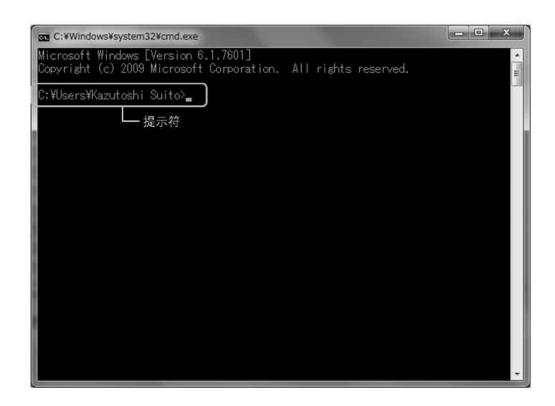


图 1-71 命令行窗口

然后我们在命令行窗口中试着执行一下 iverilog 命令。如果出现"iverilog: no source files..."信息,则没有问题。如果出现"iverilog'不是内部或外部命令,也不是可运行的程序或批处理文件。"请按照下面的步骤设定环境变量。

■ 设定命令查找路径

iverilog 无法正确执行的原因是没有正确设定命令搜索路径。命令搜索路径是 Windows 查找可执行文件的场所。 当输入 iverilog 命令时,命令行窗口会在命令搜索路径中搜索名为 iverilog 的可执行文件。iverilog 执行文件在Icarus Verilog 安装文件夹下的 bin 目录中。但是因为这个目录并未包含在命令搜索路径中,因此命令行窗口找不到执行文件。

命令搜索路径可以在环境变量中设置。环境变量是在程序执行时操作系统向应用传递的通用参数。通过设定环境变量,可以对应用的动作进行设定。环境变量设定的步骤如图 1-72 所示。

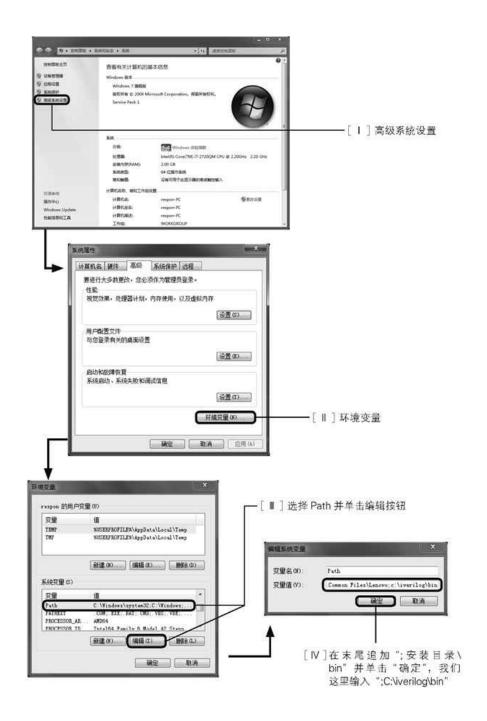


图 1-72 环境变量设定步骤

[[] 打开"计算机属性"窗口并单击"高级系统设置"。

打开"计算机属性"窗口是在开始菜单中右键单击"计算机",选择属性。

- [Ⅱ]点击"高级系统设置"中的"环境变量"。
- [III] 选择"系统变量"中的"Path",并单击编辑。

[IV] 在变量值的末尾追加";安装路径 \bin",点击"确定"。默认安装的情况下,设定字符串为";C:\iverilog\bin"。

设定好环境变量之后,再次打开命令行窗口并执行 iverilog 命令。这次应该会出现正确的输出信息 iverilog: no source files... 了。

■ 使用 Icarus Verilog 进行仿真

使用 Icarus Verilog 进行仿真,首先需要用 iverilog 命令对源代码进行编译。为 iverilog 命令的参数指定正确的选项和源代码文件,执行后就会输出编译后的文件。iverilog命令的选项如表 1-8 所示。

表 1-8	iverilog	的选项
-------	----------	-----

选项	说明
	定义macro 参数指定的宏
-D macro[=defn]	可用=defn 指定该宏的值
	如果省略,则宏的值为1
-I includedir	指定引用文件的查找路径
-o filename	指定输出文件名
-s topmodule	指定最上层模块名
-y libdir	指定库文件的查找路径

-D 选项用来定义宏。这个选项与代码中用 `define 命令定义的方式等效。通过参数设定的宏在控制仿真动作方面非常有用。-I 选项用来指定引用文件的查找路径。代码中 `include 语句引用的文件会在 -I 选项指定的目录中查找。-o 选项用来指定输出文件的文件名。省略 -o 选项时,默认输出文件名为 a.out。-s 选项用来指定最上层模块的名称。-y 选项用来指定库文件的查找路径。

编译之后,使用 vvp 命令来执行仿真。vvp 的参数中需要指定 iverilog 命令所输出的文件。vvp 命令执行后,就会按照 Testbench 中记述的测试序列进行仿真。如果 Testbench 中有波形输出,就会输出波形文件。

下面我们尝试使用 Icarus Verilog 进行仿真试验。试验用的文件为之前示例中制作的寄存器堆(regfile.v)和 Testbench(regfile_test.v)。进入代码与 Testbench 所在的目录,并在命令行中执行以下命令。

```
C:\Users\...> iverilog -s regfile_test -o regfile_test.out reg-
C:\Users\...> vvp regfile_test.out
```

iverilog 参数中指定了代码文件与 Testbench 文件。-s 参数将最上层模块名指定为 Testbench 的模块名。-o 参数设定输出文件名为 regfile_test.out。

然后执行 vvp 进行仿真,参数设置为 iverilog 的输出文件。如果仿真正确执行,画面中会出现 Testbench 中的输出信息。我们可以看到寄存器堆模块读写测试完成的信息。最后,Testbench 执行后的波形文件输出到了regfile.vcd 中。

```
C:\Users\...>vvp regfile.out

VCD info: dumpfile regfile.vcd opened for output.

475 ff[ 0] Read/Write Check OK !

675 ff[ 1] Read/Write Check OK !

.......

6475 ff[ 30] Read/Write Check OK !

6675 ff[ 31] Read/Write Check OK !
```

■ 使用 GTKWave 查看波形

下面,我们使用 GTKWave 软件查看 Icarus Verilog 输出的

波形文件。GTKWave 使用 gtkwave 命令进行启动。如果 启动时设定波形文件参数,启动后会自动载入波形文件。 GTKWave 的界面与使用方法如图 1-73 所示。

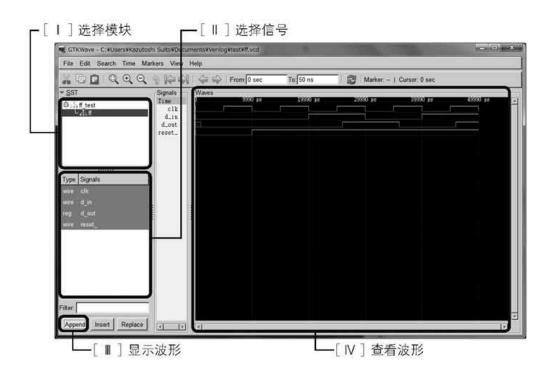


图 1-73 GTKWave 的界面与使用方法

- [I]窗口左上方会显示模块的树状列表。单击即可选择想要查看波形的模块。
- [II]窗口左侧中间会显示已选模块的信号。单击即可选择想要查看波形的信号。
- [III] 单击左下方的 **Append** 按钮,波形就会在右侧窗口中出现。
- [IV]在窗口右侧观察波形,同时可以使用滚动条或工具栏对波形的显示进行调整。

下面,我们来查看一下刚才 Icarus Verilog 输出的波形文件 regfile.vcd。进入波形文件所在的文件夹并执行以下命令,GTKWave 会启动并载入波形文件。

C:\Users\...> gtkwave regfile.vcd

1.4.5 小结

在本节中,我们介绍了 Verilog HDL 的语法、示例与仿真环境等,这是阅读接下来的章节必须掌握的基础知识。必要时,读者们可以返回查阅。

专栏

Verilog HDL 相关书籍

• 入門 Verilog HDL 記述 - ハードウェア記述言語の速習&実践(小林優、CQ出版)

(中文译名:《入门 Verilog HDL 记述·硬件描述语言速成与实践》)

本书是以 Verilog HDL 为基础的硬件设计入门书。结合实例进行讲解,通俗易懂地介绍了如何编写代码。篇幅适中、示例简洁,推荐 Verilog HDL 初学者阅读。

• LSI 設計の基本 RTL 設計スタイルガイド - Verilog HDL 編(STARC 監修、培修館)

(中文译名:《LSI设计基础 RTL 风格指南: Verilog-HDL 篇》)

本书讲解了RTL设计进阶的设计风格,阐述了设计时需要遵守的约定、代码风格等,旨在帮助正在使用Verilog HDL进行开发的读者,学习更高深的设计技术。这本书主要以具体的设计问题为中心进行讲解,难度稍高,要成为RTL设计达人,必备此书。

1.5 系统蓝图

本节将介绍本章即将制作的系统,同时也会对本章中代码的阅读方法、全局通用的宏进行说明。

1.5.1 目标系统整体介绍

AZPR SoC 是以 AZ Processor 为中心,结合存放程序的ROM(Read Only Memory)、测量时间的计时器、串口通信的 UART(Universal Asynchronous Receiver Transmitter)、控制 LED 和开关的 GPIO(General Purpose Input Output),以及连接以上模块的总线构成的。

AZ Processor 拥有专用的 SPM(Scratchpad Memory,暂时存储器),可以不通过总线进行高速访问。定时器与UART 输出的中断请求信号直接连接到 AZ Processor。AZPR SoC 还需要输入复位信号以及相位为 0 度与 180 度的两种时钟信号。基于外部输入的复位信号和基准时钟信号,时钟模块可以生成所需的复位信号与两种时钟信号。图 1-74 为本章即将制作的 AZPR SoC 的框图。

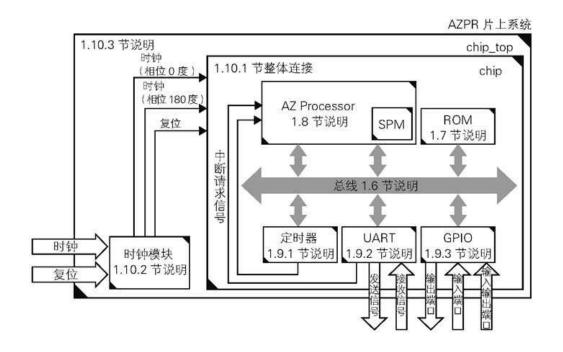


图 1-74 本章制作目标: AZPR SoC

首先,我们在 1.6 节制作用于整体通信连接的总线。其次,在 1.7 节制作 ROM。然后,在 1.8 节制作本章的主要部分——CPU。在 1.9 节制作 I/O。最后,1.10 节将各部分连接,完成 AZPR SoC。

1.5.2 关于本章中的代码

• 代码的阅读方法

本章将代码归纳到表格里进行说明。本节仅选取进行 某种控制的代码片段进行说明。我们一般会省略模块 声明与信号线定义的部分。而各个模块的端口、信号 线、头文件中定义的宏,则以列表的形式在文中给 出。

本章中, Verilog HDL 程序的一个代码文件中仅包含一个模块,并且文件名与模块名一致。表 1-9 列出了模块的层次,表 1-10 给出了头文件一览。

表 1-9 模块层次

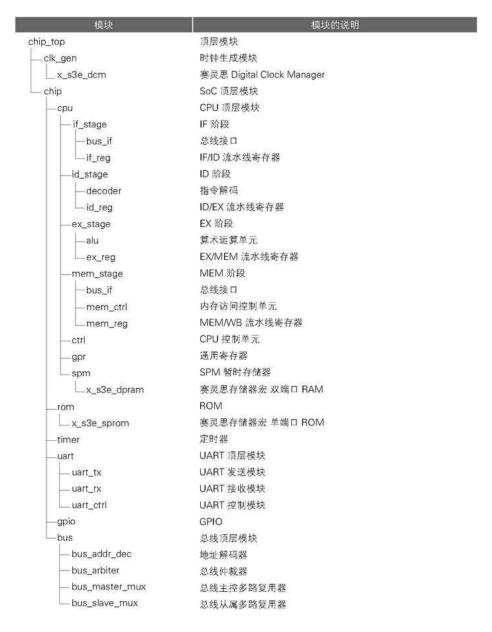


表 1-10 头文件一览

文件名	说明
nettype.h	设置默认网络类型
global_config.h	全局设置
stddef.h	通用头文件
isa.h	ISA 头文件
cpu.h	CPU 头文件
spm.h	SPM 头文件
bus.h	总线头文件

gpio.h	GPIO 头文件
rom.h	ROM 头文件
timer.h	计时器头文件
uart.h	UART 头文件

• 代码规范

本书中的 Verilog HDL 代码,以可读性和易懂性作为第一原则进行编写。为了方便读者理解,代码中尽可能插入注释进行说明。

代码中避免使用魔术数字(Magic number),而较多采用宏。魔术数字是指嵌入代码中的常数。不使用魔术数字可以增强代码的可移植性。全部宏都在头文件中定义。

每行代码文字数量都在 80 以内,行的缩进使用制表符。代码中每一行长度都控制在终端显示设备的行宽以内,这样有助于阅读。通常终端的一行可显示 80字,不单是 Verilog HDL 代码,各种代码多采用每行80字的宽度。缩进字符推荐使用制表符。制表符的优点是宽度可在文本编辑器内设定,阅读代码的人可以自由调整。笔者的环境中,一个制表符相当于 4个空格的宽度。

• 变量名与宏的命名规则

变量名使用英文小写字母、数字以及下划线(_)进行命名。为了明确控制信号的极性,负逻辑信号线的名称以下划线(_)结尾。宏使用英文大写字母、英文小写字母、数字以及下划线(_)进行命名。常数使用大写英文字母和下划线(_)进行命名。在定义比特位或总线时,使用单词首字母大写的驼峰拼写法(Upper CamelCase)。

宏的定义在头文件中进行。头文件中加入包含文件防范(Include guard)语句防止重复定义。包含文件防

范是防止同一个文件被多次包含的技术。包含文件中的代码全部写在 `ifndef 之中,并在其中定义防范用的宏。当再次引用该文件时, `ifndef 中的代码就会无效。宏的命名规则如图 1-75 所示。

```
`ifndef __INC_GUARD__// 包含文件防范`define __INC_GUARD__// 包含文件防范用的宏`define DataBus 31:0// 比特位或总线用驼峰拼写法`define DATA_W 32// 常数使用大写英文字母和下式`endif// 包含文件防范
```

图 1-75 宏的命名规则

• 全局通用宏

本章代码中,全局通用的头文件如表 1-11 所示。

表 1-11 通用头文件

文件名	作用
nettype.h	定义默认网络类型
global_config.h	定义有可能变化的参数
stddef.h	定义通用宏

nettype.h 中对 Verilog HDL 的默认网络类型进行定义。为了避免人为失误,通常将默认网络类型设置为无效。代码 1-4 为 nettype.h 的代码。

代码 1-4 定义默认网络类型(nettype.h)

```
15 `default_nettype none // none (推荐)
16 // `default_nettype wire // wire (Verilog标准)
17
18 `endif
```

global_config.h 中定义有可能变化的参数。比如说,复位信号的极性有可能随着使用端口的更换而改变,内存控制信号的极性也有可能随着 FPGA 芯片的不同而不同。这个文件还定义选择使用的 I/O 等。表 1-12 列出了 global_config.h 中的宏一览。

表 1-12 宏一览(global_config.h)

宏名	值	含义
POSITIVE_RESET	NaN	【复位信号极性的选择】
NEGATIVE_RESET	NaN	使用Active High 复位时定 义POSITIVE_RESET 使用Active Low复位时定 义NEGATIVE_RESET
POSITIVE_MEMORY	NaN	【内存控制信号极性的选
NEGATIVE_MEMORY	NaN	择】 使用Active High 复位时定 义POSITIVE_MEMORY 使用Active Low复位时定 义NEGATIVE_MEMORY
IMPLEMENT_TIMER	NaN	【I/O 的选择】
IMPLEMENT_UART	NaN	需要实现计时器时定义 IMPLEMENT TIMER
IMPLEMENT_GPIO	NaN	需要实现UART 时定义 IMPLEMENT_UART 需要实现通用I/O 时定义 IMPLEMENT_GPIO
RESET EDGE	posedge	复位信号边沿(定义 POSITIVE_RESET 时)
KEGE I_EDGE	negedge	复位信号边沿(定义 NEGATIVE_RESET 时)
DECET ENABLE	1'b1	复位有效(定义 POSITIVE_RESET 时)
RESET_ENABLE		复位有效 (定义

	1'b0	NEGATIVE_RESET 时)
RESET_DISABLE	1'b0	复位无效(定义 POSITIVE_RESET 时)
RESEI_DISABLE	1'b1	复位无效(定义 NEGATIVE_RESET 时)
MEM_ENABLE	1'b1	内存有效(定义 POSITIVE_MEMORY 时)
	1'b0	内存有效(定义 NEGATIVE_MEMORY 时)
MEM DICADI E	1'b0	内存无效(定义 POSITIVE_MEMORY 时)
MEM_DISABLE	1'b1	内存无效(定义 NEGATIVE_MEMORY 时)

stddef.h 中对全局通用宏进行定义。其中,定义了信号电平高低的 H、L,以及控制信号的有效、无效等通用宏。stddef.h 中的宏一览如表 1-13 所示。

表 1-13 宏一览 (stddef.h)

宏名	值	含义
HIGH	1'b1	高电平信号
LOW	1'b0	低电平信号
DISABLE	1'b0	无效 (正逻辑)
ENABLE	1'b1	有效 (正逻辑)
DISABLE_	1'b1	无效 (负逻辑)
ENABLE_	1'b0	有效(负逻辑)
READ	1'b1	读取信号
WRITE	1'b0	写入信号
LSB	0	最低位
BYTE_DATA_W	8	数据宽度(字节)
BYTE_MSB	7	最高位 (字节)
ByteDataBus	7:0	数据总线 (字节)
WORD_DATA_W	32	数据宽度 (字)
WORD_MSB	31	最高位 (字)

WordDataBus	31:0	数据总线 (字)
WORD_ADDR_W	30	地址宽度
WORD_ADDR_MSB	29	最高位
WordAddrBus	29:0	地址总线
BYTE_OFFSET_W	2	位移宽度
ByteOffsetBus	1:0	位移总线
WordAddrLoc	31:2	字地址位置
ByteOffsetLoc	1:0	字节位移位置
BYTE_OFFSET_WORD	2'b00	字边界

专栏

字编址与字节位移

CPU 有时需要一次处理宽度大于一个字节的数据。比如说,32 位(4 字节)CPU 需要处理 32 位数据,64 位(8 字节)CPU 需要处理 64 位数据。CPU 能处理的数据宽度称为字,为每一个字宽的数据赋予一个地址的方式称为字编址。CPU 内部因为以字为单位处理数据,方便起见,有时编址方式也采用字编址。

AZ Processor 是 32 位 CPU,一个字有 32 位(4字节)。因此每 4 个字节分配 1 个地址。AZ Processor 的寻址空间为 32 位。虽然这 32 位地址为字节编址,但 CPU 内部将高位的 30 位以字编址,低位的 2 位(4 字节的地址空间)用作字节位移使用。图 1-76 说明了字编址与字节位移的关系。



图 1-76 字编址与字节位移的关系

• 1.6 总线的设计与实现

本节介绍总线的设计与实现。总线是将 **CPU**、内存和 **I/O** 相互连接的共享通道。因为总线和本章所有电路都有关连,所以我们先来制作。

1.6.1 总线的设计

本书设计的总线主控为 4 通道,总线从属为 8 通道。总线的信号线如表 1-14 所示。

表 1-14 总线信号线

		信号方向		启	
信号名	信号名 名称		信号目 的地	位宽	含义
clk	Clock	主控、从	人属共用	1	同步信号
req_	Request	主控	总线仲 裁器	1	请求总线使用权
grnt_	Grant	总线仲 裁器	主控	1	总线使用许可信 号
addr	Address	主控	从属	30	访问地址
cs_	Chip Select	地址解 码器	从属	1	从属访问选择信 号
as_	Address Strobe	主控	从属	1	访问有效表示信 号
rw	ead/Write	主控	从属	1	访问方式(读/ 写)表示信号
wr_data	Write Data	主控	从属	32	写入数据
rd_data	Read Data	从属	主控	32	读取数据
rdy_	Ready	从属	主控	1	访问结束表示信 号

通过总线访问时,需要预先确定总线主控与总线从属 之间的通信规则。这种使用信号线的通信规则称为总 线协议。本书使用的总线为使用时钟信号同步数据传 输的同步总线。图 1-77 展示了读取访问时的总线波形。

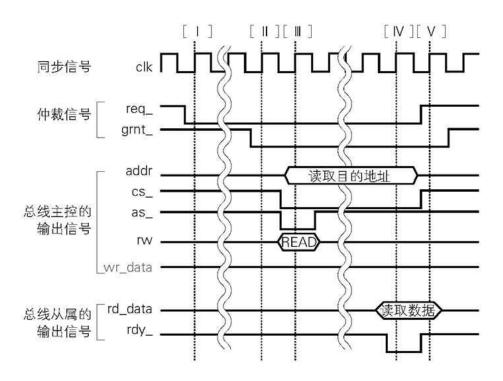


图 1-77 读取访问时的总线波形

[] 请求总线使用权

总线主控在获得总线的使用权后方可使用总线。主控发出总线使用权请求信号(req_)请求总线使用权。

[II] 取得总线使用权

总线仲裁器对总线主控发来的总线使用权请求进行调停,并发出总线使用许可信号(grnt_)。总线主控在接收到总线使用许可信号(grnt_)后,即可开始总线访问。

[III] 总线访问开始

总线主控输出地址(addr)信号,并发出地址选通(as_)信号。片选信号(cs_)由地址解码器基于地址信号生成。由于是读取访问,向读/写信号(rw)

输出读取(READ)信号。读/写信号(rw)和地址 选通(as_)保持1个时钟周期,地址(addr)信号需 要保持到总线访问结束。

[IV] 来自总线从属的应答

总线从属同时输出就绪(rdy_)信号与读取的数据(rd_data)。

「V]总线访问结束并释放总线使用权

地址(addr)信号输出停止并结束总线访问,总线使用权信号(req_)反相,释放总线使用权。

图 1-78 展示了写入访问时的总线波形。

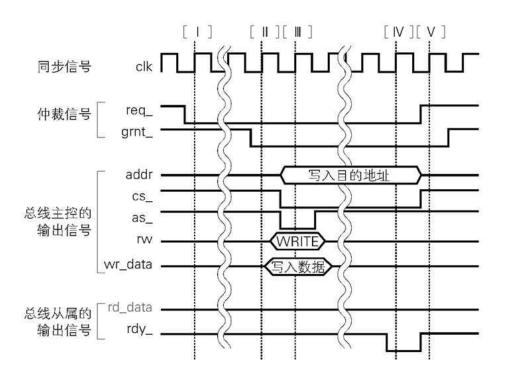


图 1-78 写入访问时的总线波形

[] 请求总线使用权

总线主控在获得总线的使用权后方可使用总线。主控发出总线使用权请求信号(req_)请求总线使用权。

「II]取得总线使用权

总线仲裁器对总线主控发来的总线使用权请求进行调停,并发出总线使用许可信号(grnt_)。总线主控在接收到总线使用许可信号(grnt_)后,即可开始总线访问。

[III] 总线访问开始

总线主控输出地址(addr)信号,并发出地址选通(as_)信号。片选信号(cs_)由地址解码器基于地址信号生成。由于是写入访问,向读/写信号(rw)输出写入(WRITE)信号,并同时输出将要写入的数据(wr_data)。读/写信号(rw)、写入数据(wr_data)以及地址选通(as_)保持1个时钟周期,地址(addr)信号需要保持到总线访问结束。

[IV] 来自总线从属的应答

总线从属输出就绪(rdy_)信号。

「V]总线访问结束并释放总线使用权

地址(addr)信号输出停止并结束总线访问,总线使用权信号(req_)设为无效,释放总线使用权。

1.6.2 总线的实现

下面讲述总线的实现。总线是由总线仲裁器、总线主控多路复用器、地址解码器以及总线从属多路复用器组成的,其中总线仲裁器调停总线使用权,总线主控多路复用器选择总线使用权所有者输出信号,地址解码器基于地址生成片选信号,总线从属多路复用器基于地址(片选信号)选择从属输出信号。表 1-15 列出了总线模块一览表,图 1-79 展示了框图,表 1-16 列出了宏一览表。

表 1-15 总线模块一览表

模块名	文件名	说明
bus	bus.v	总线顶层模块
bus_arbiter	bus_arbiter.v	总线仲裁器
bus_addr_dec	bus_addr_dec.v	地址解码器
bus_master_mux	bus_master_mux.v	总线主控用多路复用器
bus_slave_mux	bus_slave_mux.v	总线从属用多路复用器

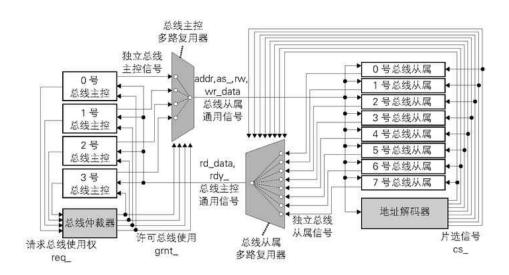


图 1-79 总线框图

表 1-16 宏一览表 (bus.h)

宏名	值	含义
BUS_MASTER_CH	4	总线主控通道数

BUS_MASTER_INDEX_W	2	总线主控索引宽度
BusOwnerBus	1:0	总线所有权状态总线
BUS_OWNER_MASTER_0	2'h0	总线使用权所有者: 0 号 总线主控
BUS_OWNER_MASTER_1	2'h1	总线使用权所有者: 1号 总线主控
BUS_OWNER_MASTER_2	2'h2	总线使用权所有者: 2 号 总线主控
BUS_OWNER_MASTER_3	2'h3	总线使用权所有者: 3号 总线主控
BUS_SLAVE_CH	8	总线从属通道数
BUS_SLAVE_INDEX_W	3	总线从属索引宽度
BusSlaveIndexBus	2:0	总线从属索引总线
BusSlaveIndexLoc	29:27	总线从属索引的位置
BUS_SLAVE_0	0	0号总线从属
BUS_SLAVE_1	1	1号总线从属
BUS_SLAVE_2	2	2号总线从属
BUS_SLAVE_3	3	3号总线从属

BUS_SLAVE_4	4	4号总线从属
BUS_SLAVE_5	5	5 号总线从属
BUS_SLAVE_6	6	6号总线从属
BUS_SLAVE_7	7	7号总线从属

图 1-79 中,各个总线主控在访问总线时,都需要向总线仲裁器发送请求信号(req_)。总线仲裁器则向被许可访问的总线主控与总线主控多路复用器输出总线使用许可信号(grnt_)。

一旦总线主控被许可使用总线,即可开始总线访问。 总线主控多路复用器基于总线许可信号(grnt_),选 择被许可访问总线的总线主控信号并输出。总线主控 多路复用器输出的信号,输入到所有总线从属和地址 解码器中。

地址解码器根据输入的地址(addr)输出片选信号(cs_)。片选信号(cs_)发送到与之对应的总线从属与总线从属多路复用器中。

总线从属多路复用器根据输入的片选信号(cs_), 选择被访问的总线从属的输出信号,并发送到总线主 控。

• 总线仲裁器的实现

总线仲裁器对总线使用权进行调停。总线仲裁器接受总线主控发来的总线使用请求,并将使用权赋予合适的总线主控。我们制作的总线仲裁器针对4个总线主控发来的请求进行调停。总线仲裁器根据目前所有者的状态,按照有限状态机方式进行控制。总线仲裁器有4个状态,分别是"0号总线主控持有总线使用权"、"1号总线主控持有

总线使用权"、"2号总线主控持有总线使用权", 以及"3号总线主控持有总线使用权"。

针对总线使用权请求的调停,使用轮询(round robin)机制。轮询是一种按照请求顺序进行使用权分配,且平等对待所有总线主控的机制。

0号总线主控拥有总线使用权时,总线请求优先级顺序为"0号总线主控>1号总线主控>2号总线主控>3号总线主控"。也就是说,0号总线主控如果要求继续使用总线,就会得到许可。

0号总线主控释放总线,1号总线主控请求使用总线时,无论2号和3号总线主控是否有请求,都会将总线使用权赋予1号。如果0号总线主控释放总线,1号总线主控没有请求使用总线,而2号总线主控请求使用总线时,则无论3号总线主控是否有请求,都会将总线使用权赋予2号。如果0号总线主控释放总线,1号与2号总线主控都没有请求使用总线,而3号总线主控请求使用总线时,3号总线主控就会获得总线使用权。最后,如果所有总线主控都没有请求使用总线,则会保持当前状态。

同样,当1号总线主控持有总线使用权时,优先级顺序为"1号总线主控>2号总线主控>3号总线主控>0号总线主控";当2号总线主控持有总线使用权时,优先级顺序为"2号总线主控>3号总线主控>0号总线主控>1号总线主控";当3号总线主控持有总线使用权时,优先级顺序为"3号总线主控"。由于每当总线主控变化时总线使用权优先级按环状变化,所有总线主控都会平等获取总线使用权。总线仲裁器的状态转移图如图1-80所示。

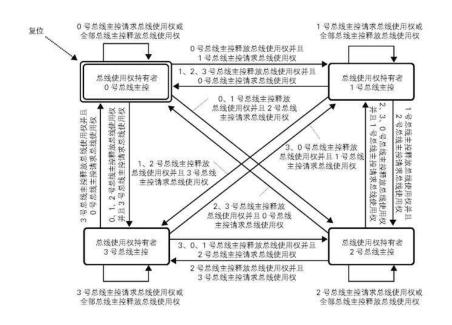


图 1-80 总线仲裁器的状态转移图

bus_arbiter 的信号线一览如表 1-17 所示。总线使用权请求信号称为 Bus request,总线使用权许可信号称为 Bus grant。代码 1-5 是生成 Bus grant信号的部分代码。

表 1-17 信号线一览(bus_arbiter.v)

分组	信号名	信号 类型	数据类 型	位 宽	含义
时钟复	clk	输入 端口	wire	1	时钟
位	reset	输入 端口	wire	1	异步复位
0 号总	m0_req_	输入 端口	wire	1	请求总线
线主控	m0_grnt_	输出 端口	reg	1	赋予总线
1 号总	m1_req_	输入 端口	wire	1	请求总线
线主控	m1_grnt_	输出 端口	reg	1	赋予总线
	m2_req_	输入	wire	1	请求总线

2号总		端口			
线主控	m2_grnt_	输出端口	reg	1	赋予总线
3 号总	m3_req_	输入 端口	wire	1	请求总线
线主控	m3_grnt_	输出端口	reg	1	赋予总线
内部信号	owner	内部 信号	reg	2	总线使用权 所有者

代码 1-5 总线仲裁器的赋予总线使用权部分 (bus_arbiter.v)

```
/****** 赋予总线使用权 *******/
43
      always @(*) begin
       /* 赋予总线使用权的初始化 */
44
         m0_grnt_ = `DISABLE_;
m1_grnt_ = `DISABLE_;
45
46
                                             一[1]赋予总线使用权的初始化
       m2_grnt_ = 'DISABLE_;
m3_grnt_ = 'DISABLE_;
47
48
                                             - [ || ] 赋予总线使用权
49
          /* 赋予总线使用权 */
     case (owner)
51
              `BUS OWNER MASTER 0 : begin // 0号总线主控
                 m0_grnt_ = `ENABLE_;
                                                     (1)0号总线主控
52
53
              end
              `BUS_OWNER_MASTER_1 : begin // 1号总线主控
54
55
                 m1_grnt_ = "ENABLE_;
                                                     (2)1号总线主控
56
              `BUS_OWNER_MASTER_2 : begin // 2号总线主控
57
                 m2_grnt_ = `ENABLE_;
                                                     (3)2号总线主控
59
              `BUS_OWNER_MASTER_3 : begin // 3号总线主控
60
                 m3_grnt_ = `ENABLE_;
                                                     (4)3号总线主控
61
62
              end
63
          endcase
```

[[] 赋予总线使用权的初始化

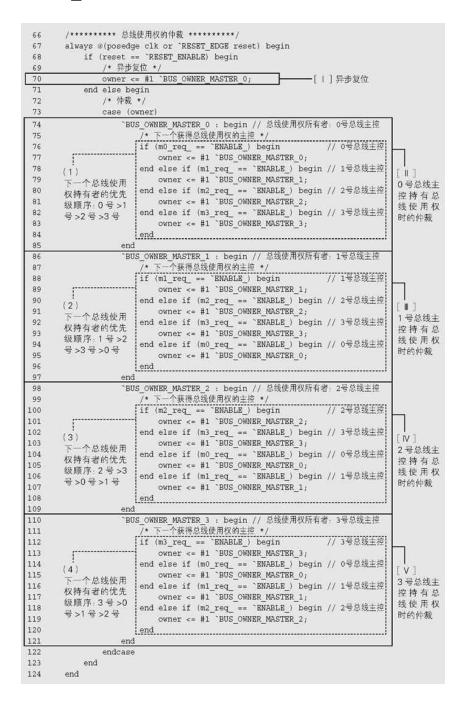
首先初始化,将所有总线主控的总线赋予信号设为无效。

[II] 赋予总线使用权

基于总线使用权持有者(owner),设置当前总线持有者信号。(1)为0号总线主控持有总线使用权的情况;(2)为1号总线主控持有总线使用权的情况;(3)为2号总线主控持有总线使用权的情况;(4)为3号总线主控持有总线使用权的情况。

总线仲裁部分的程序如代码 1-6 所示。

代码 **1-6** 总线仲裁器的仲裁部分(**bus_arbiter.v**)



[I] 异步复位

将总线使用权的持有者复位。复位后 0 号总线主 控持有总线使用权。

[Ⅱ]**0**号总线主控持有总线使用权时的仲裁

(1) 处决定下一个获取总线使用权的主控。当前主控为 0 号总线主控时,总线使用权分配优先级顺序为"0 号总线主控 >1 号总线主控 >2 号总线主控 >3 号总线主控"。没有总线使用权请求时维持当前状态。

[III] 1号总线主控持有总线使用权时的仲裁

(2) 处决定下一个获取总线使用权的主控。当前主控为1号总线主控时,总线使用权分配优先级顺序为"1号总线主控 >2号总线主控 >3号总线主控 >0号总线主控"。

[IV] 2号总线主控持有总线使用权时的仲裁

(3) 处决定下一个获取总线使用权的主控。当前主控为 2 号总线主控时,总线使用权分配优先级顺序为"2 号总线主控 >3 号总线主控 >0 号总线主控 >1 号总线主控"。

[V] 3 号总线主控持有总线使用权时的仲裁

(4) 处决定下一个获取总线使用权的主控。当前主控为3号总线主控时,总线使用权分配优先级顺序为"3号总线主控>0号总线主控>1号总线主控>2号总线主控"。

• 总线主控多路复用器的实现

总线主控多路复用器基于总线仲裁器输出的总线赋予信号,选择总线使用权所有者的信号,并将其输出到总线。bus_master_mux的信号一览如表1-18所示,示例程序如代码1-7所示。

表 1-18 信号一览表(bus_master_mux.v)

分组	信号名	信号 类型	数据 类型	位宽	含义
	m0_addr	输入 端口	wire	30	地址
	m0_as_	输入 端口	wire	1	地址选通
0号总线主控	m0_rw	输入 端口	wire	1	读/写
	m0_wr_data	输入 端口	wire	32	写入的数 据
	m0_grnt_	输入 端口	wire	1	赋予总线
	m1_addr	输入 端口	wire	30	地址
	m1_as_	输入 端口	wire	1	地址选通
1号总 线主控	m1_rw	输入 端口	wire	1	读/写
	m1_wr_data	输入 端口	wire	32	写入的数 据
	m1_grnt_	输入 端口	wire	1	赋予总线
	m2_addr	输入 端口	wire	30	地址
	m2_as_	输入 端口	wire	1	地址选通
2号总线主控	m2_rw	输入 端口	wire	1	读/写
	m2_wr_data	输入 端口	wire	32	写入的数 据
	m2_grnt_	输入 端口	wire	1	赋予总线
	m3_addr	输入 端口	wire	30	地址
	m3_as_	输入 端口	wire	1	地址选通
3号总 线主控	m3_rw	输入 端口	wire	1	读/ 写

	m3_wr_data	输入 端口	wire	32	写入的数 据
	m3_grnt_	输入 端口	wire	1	赋予总线
	s_addr	输出 端口	reg	30	地址
共享信 号	s_as_	输出 端口	reg	1	地址选通
总线从属	s_rw	输出 端口	reg	1	读/ 写
	s_wr_data	输出 端口	reg	32	写入的数 据

代码 1-7 总线主控多路复用器 (bus_master_mux.v)

```
53
        /****** 总线主控多路复用器 *******/
       always @(*) begin
54
                                               [ | ]选择持有总线使用权的主控。
            57
                                                        (1)选择0号总线主控的信号
58
59
            s wr data = m0 wr data;
end else if (ml_grnt_ == `ENABLB_) begin // 1号总线主控
60
61
             s_addr = ml_addr;
s_as_ = ml_as_;
s_rw = ml_rw;
62
63
                                                        (2)选择1号总线主控的信号
            ____swr_data = m1_wr_data;
end else if (m2_grnt_ == `ENABLB_) begin // 2号总线主控
65
66
             s_addr = m2_addr;
s_as_ = m2_as_;
s_rw = m2_rw;
67
                                                        (3)选择2号总线主控的信号
68
69
            s_wr_data = m2_wr_data;
end else if (m3_grnt_ == ENABLE_) begin // 3号总线主控
70
            s_addr = m3_addr;
s_as_ = m3_as_;
73
                                                        (4)选择3号总线主控的信号
74
               s_rw
                          = m3_rw;
            s wr data = m3 wr data;
end else begin
75
                                                       // 默认值
76
               s_addr = `WORD_ADDR_W'h0;
s_as_ = `DISABLE_;
s_rw = `READ;
77
                                                        (5)默认值
78
79
                s_wr_data = `WORD_DATA_W'h0;
80
81
        end
```

[I] 选择持有总线使用权的主控

- (1)处为当0号总线主控持有总线使用权时,将0号总线主控的信号输出到总线;同样地,
- (2) 处为当1号总线主控持有总线使用权的情况;(3)处为当2号总线主控持有总线使用权

的情况; (4) 处为当 3 号总线主控持有总线使用权的情况; (5) 处设定默认值并输出到总线。

• 地址解码器的实现

地址解码器基于总线主控输出的地址信号,判断将要访问哪个总线从属,并生成片选信号。访问的地址与总线从属的对应关系称为地址映射(address map)。

因为本书设计的总线连接到 8 个总线从属通道, 所以单纯地将地址空间 8 等分,并分配给 0 号总 线从属到 7 号总线从属。表 1-19 列出了地址映 射关系。

表 1-19 总线的地址映射

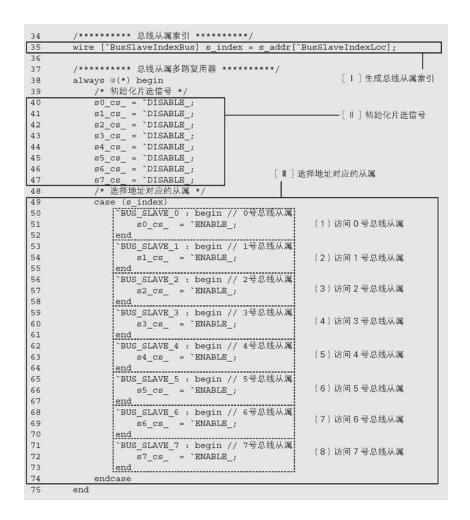
总线从 属	地址	地址最 高 3 位	分配
0号	0 x0000_0000 \sim 0x1FFF_FFFF	3'b000	只读存储 器 ROM
1号	0x2000_0000 ~ 0x3FFF_FFFF	3'b001	暂时存储 器 SPM
2号	0x4000_0000 ~ 0x5FFF_FFFF	3'b010	计时器
3号	0x6000_0000 ~ 0x7FFF_FFFF	3'b011	UART
4号	0x8000_0000 ~ 0x9FFF_FFFF	3'b100	GPIO
5号	0xA000_0000 ~ 0xBFFF_FFFF	3'b101	未分配
6号	0xC000_0000 ~ 0xDFFF_FFFF	3'b110	未分配
7号	0xE000_0000 ~ 0xFFFF_FFFF	3'b111	未分配

bus_addr_dec 的信号一览如表 1-20 所示,程序如 代码 1-8 所示。

表 1-20 信号一览表(bus_addr_dec)

分组	信号名	信号类 型	数据类型	位 宽	含义
总线从属共 享信号	s_addr	输入端口	wire	30	地址
0号总线从属	s0_cs_	输出端口	reg	1	片选
1号总线从属	s1_cs_	输出端口	reg	1	片选
2号总线从属	s2_cs_	输出端口	reg	1	片选
3号总线从属	s3_cs_	输出端口	reg	1	片选
4号总线从属	s4_cs_	输出端口	reg	1	片选
5号总线从属	s5_cs_	输出端口	reg	1	片选
6 号总线从属	s6_cs_	输出端口	reg	1	片选
7号总线从属	s7_cs_	输出端口	reg	1	片选
内部信号	s_index	内部信 号	wire	3	总线从属 的索引

代码 1-8 地址解码器 (bus_addr_dec.v)



[I] 生成总线从属索引

因为需要 3 个比特位(2 的 3 次方为 8)来区分 8 个总线从属通道,所以地址的最高 3 位用来识别总线从属。并且基于地址(s_addr)的最高 3 位生成总线从属索引(s_index)。

[II] 初始化片选信号

初始化时设置所有片选信号无效。

「III】选择总线从属索引对应的从属

对总线从属索引对应的从属发送片选信号。

(1) 处索引为0时,对0号总线从属发送片选信号。同样地,向从属发送片选信号的代码为:

1号总线从属为(2)处,2号总线从属为(3)处,3号总线从属为(4)处,4号总线从属为(5)处,5号总线从属为(6)处,6号总线从属为(7)处,7号总线从属为(8)处。

• 总线从属多路复用器的实现

总线从属多路复用器基于地址解码器输出的片选信号,将被选择的总线从属的输出信号发送到总线。bus_slave_mux的信号一览如表 1-21 所示,程序如代码 1-9 所示。

表 1-21 信号一览(bus_slave_mux.v)

	1	T	ı		T
分组	信号名	信号 类型	数据类型	位 宽	含义
	s0_cs_	输入 端口	wire	1	片选
0 号总线 从属	s0_rd_data	输入 端口	wire	32	读出的数据
	s0_rdy_	输入 端口	wire	1	就绪
	s1_cs_	输入 端口	wire	1	片选
1号总线 从属	s1_rd_data	输入 端口	wire	32	读出的数据
	s1_rdy_	输入 端口	wire	1	就绪
	s2_cs_	输入 端口	wire	1	片选
2 号总线 从属	s2_rd_data	输入 端口	wire	32	读出的数据
	s2_rdy_	输入 端口	wire	1	就绪
	s3_cs_	输入 端口	wire	1	片选
3 号总线 从属	s3_rd_data	输入 端口	wire	32	读出的数据
	s3_rdy_	输入	wire	1	就绪

		端口			
	s4_cs_	输入 端口	wire	1	片选
4号总线 从属	s4_rd_data	输入 端口	wire	32	读出的数据
	s4_rdy_	输入 端口	wire	1	就绪
	s5_cs_	输入 端口	wire	1	片选
5 号总线 从属	s5_rd_data	输入 端口	wire	32	读出的数据
	s5_rdy_	输入 端口	wire	1	就绪
	s6_cs_	输入 端口	wire	1	片选
6 号总线 从属	s6_rd_data	输入 端口	wire	32	读出的数据
	s6_rdy_	输入 端口	wire	1	就绪
	s7_cs_	输入 端口	wire	1	片选
7号总线 从属	s7_rd_data	输入 端口	wire	32	读出的数据
	s7_rdy_	输入 端口	wire	1	就绪
总线主控	m_rd_data	输出 端口	reg	32	读出的数据
共享信号	m_rdy_	输出 端口	reg	1	就绪

代码 **1-9** 总线从属多路复用器(**bus_slave_mux.v**)

61	always @(*) begin	「1]选择片选信号对应的从
62	/* 选择片选信号对应的从属 */	
63	if (s0 cs == `ENABLE) begin // 052	[野从屋]
64	m rd data = s0 rd data;	(1)访问0号总线从属
65	m rdy = s0 rdy;	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
66	end else if (sl cs == ENABLE) begin // 1号尼	(現从黨)
67	m rd data = s1 rd data;	(2)访问1号总线从属
68	m rdy = sl rdy ;	
69	end else if (s2 cs == ENABLE) begin // 2号总	(我从魔
70	m_rd_data = s2_rd_data;	(3)访问2号总线从属
71	m_rdy_ = s2_rdy_;	
72	end else if (s3_cs_ == `ENABLE_) begin // 3号总	(我从属
73	m_rd_data = s3_rd_data;	(4)访问3号总线从属
74	m_rdy_ = s3_rdy_;	i i
75	end else if (s4_cs_ == `ENABLE_) begin // 4号总	(我从属
76	m_rd_data = s4_rd_data;	(5)访问4号总线从属
77	m_rdy_ = s4_rdy_;	
78	end else if (s5_cs_ == `ENABLE_) begin // 5号总	(残从魔)
79	m_rd_data = s5_rd_data;	(6)访问5号总线从属
80	m_rdy_ = s5_rdy_;	
81	end else if (s6_cs_ == `ENABLE_) begin // 6号문	
82	m_rd_data = s6_rd_data;	(7)访问6号总线从属
83	m_rdy_ = s6_rdy_;	
84	end else if (s7_cs_ == `ENABLE_) begin // 7号总	
85	m_rd_data = s7_rd_data;	(8)访问7号总线从属
86	m_rdy = s7_rdy; end else begin // 默汉	
87		ii i
88	m_rd_data = `WORD_DATA_W'h0;	(9) 默认值
89	m_rdy_ = `DISABLE_;	
90	end	S. CARROLLONI

[1] 选择片选信号对应的从属

总线从属的选择是通过片选信号实现的。(1) 处访问 0 号总线从属时,将来自 0 号总线从属输 出的 s0_rd_data 与 s0_rdy_ 发送到总线;同样 地,(2)处为 1 号总线从属的访问;(3)处为 2 号总线从属的访问;(4)处为 3 号总线从属 的访问;(5)处为 4 号总线从属的访问;(6) 处为 5 号总线从属的访问;(7)处为 6 号总线 从属的访问;(8)处为 7 号总线从属的访问; (9)处为默认值的设定。

• 总线的顶层模块

最后对总线的顶层模块进行说明。总线的顶层模块是将总线仲裁器、总线主控多路复用器、地址解码器以及总线从属多路复用器 4 个模块进行连接的模块。

图 1-81 展示了各个模块与信号线的连接图。总线主控独立信号为 4 通道,总线从属独立信号为

8 通道。为了避免图示变得繁杂,图 1-81 中这两个通道以 [0..3][0..7] 的方式书写表示。

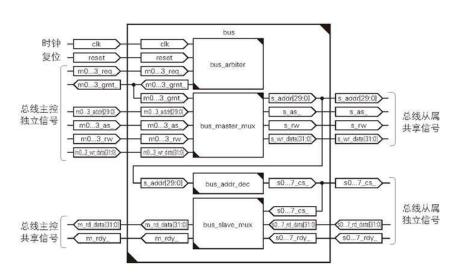


图 1-81 总线顶层模块的连接图

1.6.3 小结

本节对总线的设计与实现进行了说明。这里介绍的是经典的总线结构,经过我们实际动手设计与实现,大家应该已经对总线上数据交换的过程有了更深入的理解。

1.7 存储器的设计与实现

本节将介绍存放数据和程序的存储器的设计与实现。制作存储器用到了 FPGA 的 RAM 区域。

1.7.1 FPGA 的 RAM 区域

许多 FPGA 都有可供用户自由使用的 RAM 区域。赛灵思生产的 FPGA 称之为块 RAM,大小从几千字节到几兆字节不等。在第 2 章将要设计的电路板上搭载的 Spartan-3E XC3S250E,有27KB 可以利用的块 RAM。

块 RAM 可以作为子模块,以实例化的方式使用。块 RAM 提供的功能如表 1-22 所示。本书使用 Single Port ROM 和 True Dual Port RAM 两种类型。

表 1-22 块 RAM 的种类

种类	说明
Single Port RAM	读写使用同一端口的单端口 RAM
Simple Dual Port RAM	一个写入端口,一个读取端口的双端 口 RAM
True Dual Port RAM	两个读写端口的双端口 RAM
Single Port ROM	一个读取端口的单端口 ROM

Dual Port ROM	两个读取端口的双端口 ROM

更多关于赛灵思的块 RAM 的资料请参阅下面的连接。这里,我们仅对本书使用的功能进行说明。

Using Block RAM in Spartan-3 Generation FPGAs

http://www.xilinx.com/support/documentation/applic

• Single Port ROM

Single Port ROM 是单一端口读取专用的存储器。Single Port ROM 的输入输出端口如表 1-23 所示,访问时序图如图 1-82 所示。模块名、存储区域宽度和深度等参数在实例化时再决定。

表 1-23 Single Port ROM 的输入输出端口

分组	信号名	信号类型	位宽	含义
A端口	clka	输入	1	时钟
	addra	输入	实例化 时决定	地址
	douta	输出	实例化 时决定	读取的数据

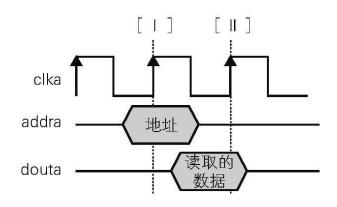


图 1-82 Single Port ROM 的访问时序

[I] 锁存输入的地址

时钟信号(clka)上升沿时将地址(addra)锁存。地址(addra)锁存后输出读取的数据(douta)。

[II] 锁存输出的数据

地址(addra)锁存后的下一个时钟周期,即可将读取的数据(douta)锁存。

• True Dual Port RAM

True Dual Port RAM 是双端口读写存储器。 True Dual Port RAM 的两个端口可以同时访问。各个端口可以有独立的时钟。True Dual Port RAM 的输入输出端口如表 1-24 所示, 访问时序图如图 1-83 所示。模块名、存储 区域宽度和深度等参数在实例化时再决定。

表 **1-24** True Dual Port RAM 的输入输出 端口

分组	信号名	信号类型	位宽	含义
	clka	输入	1	时钟
	wea	输入	实例化时决 定	写入使能

A端口	addra	输入	实例化时决定	地址	
	dina	输入	实例化时决定	写入的数 据	
	douta	输出	实例化时决 定	读取的数 据	
B端口	clkb	输入	1	时钟	
	web	输入	实例化时决 定	写入使能	
	addrb	输入	实例化时决定	地址	
	dinb	输入	实例化时决 定	写入的数 据	
	doutb	输出	实例化时决定	读取的数 据	

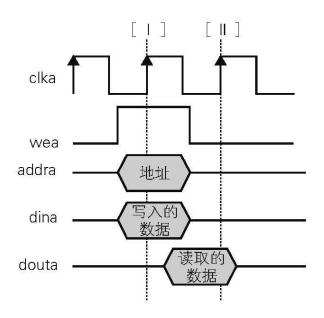


图 1-83 Dual Port RAM 的访问时序图

[] 锁存输入的地址

时钟信号(clka)上升沿时将地址(addra)锁存。此时,如果写入使能信号(wea)有效,则将写入的数据(dina)写入存储器。地址(addra)锁存后输出读取的数据

(douta) .

[II] 锁存输出的数据

地址(addra)锁存后的下一个时钟周期,即可将读取的数据(douta)锁存。

由于 True Dual Port RAM 可以同时在两个端口进行读写操作,因此在两个端口同时对相同地址进行读写访问时应加以注意。此时的操作可以在块 RAM 实例化时加以设置。在此不多做介绍,详情请参阅前文提到的块RAM 文档。

1.7.2 ROM 的设计与实现

本节设计的 ROM 将用来存放引导程序。ROM 地址映射到地址 0 处,AZ Processor 启动后从 0 号地址开始执行程序。ROM 由单个名为 rom 的模块构成。存储器使用 Single Port ROM。表 1-25 列出了 rom 模块使用的宏一览,表 1-26 列出了 rom 模块信号线一览,代码 1-10 列出了 rom 模块的程序。

表 1-25 宏一览 (rom.h)

宏名	值	含义
ROM_SIZE	8192	ROM 的大小
ROM_DEPTH	2048	ROM 的深度
ROM_ADDR_W	11	地址宽度
RomAddrBus	10:0	地址总线

RomAddrLoc	10:0	地址的位置
------------	------	-------

表 1-26 信号线一览 (rom.v)

分组	信号名	信号类型	数据类型	位宽	含义
时钟 / 复位	clk	输入端口	wire	1	时钟
	reset	输入端口	wire	1	异步复位
	cs_	输入端口	wire	1	片选信号
	as_	输入端口	wire	1	地址选通
总线接口	addr	输入端口	wire	11	地址
	rd_data	输出端口	wire	32	读取的数 据
	rdy_	输出端口	reg	1	就绪信号

代码 1-10 Read Only Memory (rom.v)

```
/****** Xilinx FPGA Block RAM : 单端口ROM ********/
33
        x_s3e_sprom x_s3e_sprom (
                                              // 时钟
34
            .clka (clk),
                                              // 地址
35
             .addra (addr),
                                              // 读取的数据
36
             .douta (rd_data)
37
38
                                                         —

- [ Ⅰ ] 存储器的实例化
         /****** 生成就绪信号 *******/
39
        always @(posedge clk or `RESET_EDGE reset) begin
40
            if (reset == `RESET_ENABLE) begin
/* 异步复位 */
41
42
                                                           [ || ] 异步复位
                rdy_ <= #1 `DISABLE_;
44
             end else begin
                 /* 生成就绪信号 */
45
                if ((cs == `ENABLE_) && (as_ == `ENABLE_)) begin rdy_ <= #1 `ENABLE_; end else begin
46
47
48
                    rdy_ <= #1 `DISABLE_;
49
50
                                                              - [ ▮] 生成就绪信号
        end
```

[I] 存储器的实例化

块 RAM 的实例化。

[II]异步复位

复位信号有效时,将就绪信号初始化。

[III] 生成就绪信号

片选信号与地址选通信号同时有效时,因为即将 访问总线,就绪信号变为有效。其他情况时,就 绪信号设置为无效。

1.7.3 小结

本节讲解了存储器的设计与实现。主要描述了 FPGA 的块 RAM 的使用方法。

专栏

存储器相关书籍

メモリ IC の実践活用法(桑野雅彦 著、CQ 出版)

(中文译名:《存储器 IC 的实践与活用方法》)

本书通俗易懂地讲述了各种存储器的构造、原理以及使用方法。通过阅读本书,读者们除了可以掌握存储器的基础知识,还可以了解从事电路设计所必需的存储器知识。

1.8 AZ Processor 的设计与实现

本节将对 **CPU** 的设计与实现进行说明。首先讲述流水线处理技术的概要和实现方法,然后设计并实现 **AZ Processor**。

1.8.1 关于 CPU

• 流水线处理

流水线处理是一种提高 CPU 的处理性能的 技术。所谓流水线处理,是将处理操作分为 多个阶段,然后像流水线作业一样执行。图 1-84 展示了流水线处理的示意图。

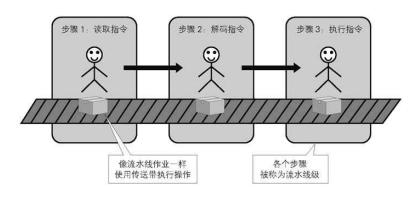


图 1-84 流水线处理示意图

CPU 中的各种硬件资源,只在处理的相应阶段使用,其他时间大多处于空闲状态。比如,运算器在指令执行时使用,在指令读取、解码时空闲。因此,为了高效使用这些硬件资源,引入了流水线处理技术。

在流水线处理的情况下,读取某条指令之后,在该指令解码的同时读取下一条指令。 通过使各个阶段的动作重叠,可以让硬件资 源有效使用,同时提高处理速度。流水线处理就像是将之前一个人完成的操作,分成 N 个相连的步骤进行处理,以此将处理效率提高 N 倍。

流水线处理中,处理的各个阶段被称为流水线级。各个流水线级的处理时间应该尽量相等。因为如果各个流水线级的处理时间不均等的话,最慢的流水线级的处理时间将成为系统的时钟周期。因此,多数 CPU 会进一步细化读取、解码和执行这 3 个步骤,以实现高效的流水线。

最为典型的流水线分为 5 个阶段,请参见图 1-85。使用了流水线技术的 CPU,通常在各个流水线级之间设置流水线寄存器,用来保存状态并传递给下一个操作阶段。

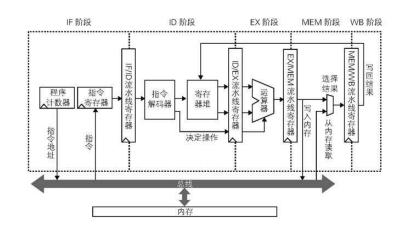


图 1-85 CPU 的流水线化

IF (Instruction Fetch) 阶段

将 PC 的值发送到内存,读取指令。

ID(Instruction Decode)阶段

将读取的指令解码并决定将要进行的操作,

从寄存器堆读取数据。

EX(Execution)阶段

使用运算器执行操作。可以执行算术运算和逻辑运算的运算器称为 ALU(Arithmetic and Logic Unit)。

MEM (Memory Access) 阶段

进行内存访问。

WB(Write Back)阶段

将结果写回寄存器堆。

实现了流水线化的 CPU,将这 5 个流水线级的操作重叠使用,按照图 1-86 所示的方式执行。

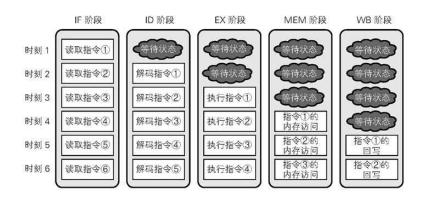


图 1-86 流水线的流程

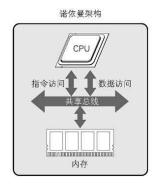
• 流水线冒险

流水线处理中,由于各个阶段的依赖关系、硬件资源的竞争等原因,会出现操作无法执行的情况。造成流水线故障的原因称为冒险,冒险分为构造、数据冒险和控制冒险 3 种类型。

• 构造冒险

构造冒险是指由于硬件资源的竞争,操作无法同时执行的情况。图 1-85 所示的流水线结构中,内存访问会造成构改冒险。IF 阶段和 MEM 阶段都要涉及内存访问。由于访问内存使用的总线是共享资源,无法同时进行操作。因此,如果发生 IF 阶段和 MEM 阶段同时访问内存的情况,一方需要等待另一方访问完成。这种指令和数据使用同一通道的构造称为诺依曼架构。

如果导致冒险产生的硬件资源数量足够多,也可以避免冒险问题的发生。因此,指令用的内存和数据用的内存分别设置,即可解决构造冒险的问题。这种物理上将指令用和数据用的内存与访问通道分开的构造称为哈佛架构。图 1-87 分别展示了诺依曼架构和哈佛架构。



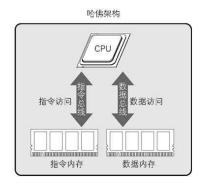


图 1-87 诺依曼架构与哈佛架构

哈佛架构的优点是,就算指令访问和数据访问同时发生,也不会发生冒险的情况。但是,也有指令和数据地址空间不同的缺点。在哈佛架构中,指令的0号地址和数据的0号地址指向不同的内容。这会引起软件设计上的问题。

近年来,大部分 CPU 的指令和数据都放在同一内存中。但是,CPU 直接访问的缓存基本上都分为指令用和数据用两种,称为指令缓存和数据缓存。图 1-88 展示了带有缓存的 CPU 构造。通过两种缓存的使用,解决了指令访问和数据访问之间发生的构造冒险问题。

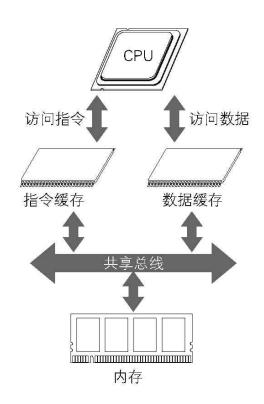


图 1-88 带有缓存的 CPU 构造

• 数据冒险

数据冒险是指,由于指令执行所需要的数据还未准备好所引起的冒险情况。当即将执行的指令依赖于还未处理完成的数据时,会导致指令无法立刻开始执行,引发数据冒险。

为了回避数据冒险,我们使用一种称为 直通(Forwarding)的方法。原本回写 运算结果是在 WB 阶段,但实际上决定 运算结果是在 EX 阶段。因此直通是指 在运算结果确定的 EX 阶段,将数据直 接传递给下一个指令。

直通的示例如图 1-89 所示。示例中使用流水线执行 3 条有数据依赖关系的指令,以此说明直通的动作原理。第二条指令要使用第一条指令的结果。第一条指令在 EX 阶段可以确定运算结果后,直接将结果发送到处于 ID 阶段的第二条指令。第三条指令同时依赖于第一条指令。第二条指令。因此,可以直接从处于 EX 阶段的第二条指令获取数据。以这种将运算结果直通的方式,可以消除原本需要等待 WB 阶段完成的依赖关系。

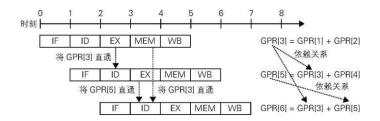


图 1-89 直通示例

使用直通解决依赖关系的方法仅有一个例外,就是数据需要使用 Load 指令从内存调取的情况。由于内存的访问在MEM 阶段执行,因此处理结果要在MEM 阶段才能确定。而当前指令执行到 MEM 结束时,下一条指令已经到达EX 阶段执行了。这与直通的机制不吻合。

这种依赖 Load 指令而发生的冒险称为 Load 冒险。Load 冒险不能从根本上避 免,因此要将有依赖关系的指令进行阻 塞以解决该问题。阻塞是指让流水线的 特定阶段停止一段时间。Load 冒险发生时的流水线动作如图 1-90 所示。

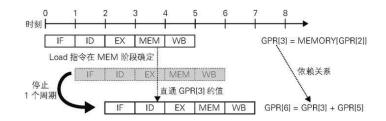


图 **1-90** Load 冒险发生时的流水线的动作

如果有 Load 冒险发生,则将有依赖关系的指令延迟一个周期执行。如果将指令阻塞一个周期,前一条指令在 MEM 阶段得到的数据就可以直通正在 ID 阶段的下一条指令。这时候,流水线会浪费一个周期,这一周期让其传递无效的数据即可。这个操作称为流水线冒泡。如果 Load 指令与和其有依赖关系的指令相差一条以上指令的距离,则不会发生 Load 冒险。作为有效的处理操作的方法,在编译器中使用适当的调度算法也可以有效避免 Load 冒险。

• 控制冒险

控制冒险是指无法确定下一条指令而引发的冒险情况。在执行可能会改变下一条指令的分支指令时,在这一条指令执行结果确定之前下一条指令无法开始执行,从而引起控制冒险。

控制冒险也无法从根本上避免,但是可以尽量将分支指令安排到流水线前段,从而减少因为控制冒险而引起的无效指令数量。比如在 ID 阶段判定分支后,延迟一个周期就可以开始执行分支指向

的下一条指令。控制冒险发生时的流水 线动作如图 1-91 所示。

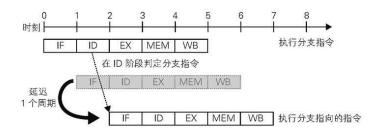


图 1-91 控制冒险发生时的流水线动作

因为在读取下一条指令前需要确定 PC 寄存器的值,即使在 ID 阶段判定分支也会产生一个周期的延迟。延迟期间会让流水线传送无效数据。流水线冒泡会浪费硬件资源,因此可以采用延迟分支的方法,许可分支指令的下一条指令执行。

延迟分支是指分支指令执行后并不立刻 跳转到分支结果指向的指令,在分支指 令的下一条指令执行完毕后再进行跳 转。分支指令的下一条指令称为延迟间 隙,不论分支是否成立都会被执行。使 用延迟分支可以避免流水线冒泡,使操 作的处理更有效率。一般的分支与延迟 分支如图 1-92 所示,采用了延迟分支 的流水线执行过程如图 1-93 所示。

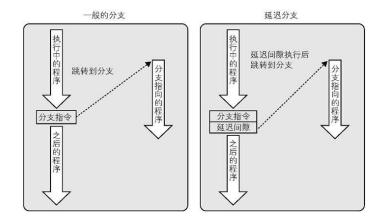


图 1-92 一般的分支与延迟分支

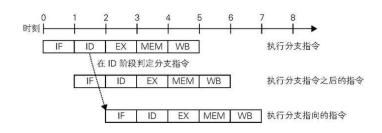


图 1-93 采用了延迟分支的流水线执行过程

• CPU 模式

大部分 CPU 至少都有两种 CPU 模式。CPU 模式也称为特权级,它会根据 CPU 的工作 模式限制可以执行的操作。CPU 模式中,全 部指令可以无限制执行的模式称为内核模式 (Kernel Mode) 或管理者模式(Supervisor Mode),操作系统等系统软件需要在内核 模式下工作。反之,可执行的指令被限制的 模式称为用户模式(User Mode),应用软 件通常在用户模式下工作。用户模式中被限 制的操作包括 CPU 控制寄存器的访问、改 变 CPU 状态的指令等。如果应用程序擅自 更改 CPU 的状态,最坏会导致操作系统崩 溃。因此,需要根据 CPU 模式管理各种软 件的权限。 大多情况下,CPU 的控制寄存器内都有可以设置 CPU 模式的区域。在从高权限的内核模式转换到低权限的用户模式时,可以通过操作控制寄存器来实现。反之,如果要从低权限的用户模式转换到高权限的内核模式,需要使用专用的指令。

• 中断和异常

中断是指让 CPU 暂停正在执行的操作,执行其他操作的功能。中断经常用在通知来自 I/O 的事件、处理程序执行中的异步事件等情况。发生中断时,CPU 暂停当前操作,并跳转到中断处理程序。这时,CPU 模式会变更到内核模式。中断处理完成后返回到中断处继续执行。中断处理的流程如图 1-94 所示。

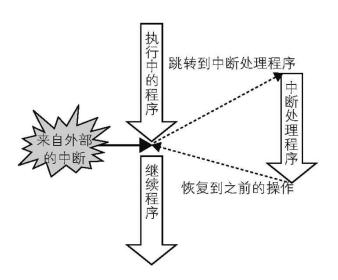


图 1-94 中断处理的流程

异常是指 CPU 的执行产生了预期之外的结果。例如,遇到无法解码的指令、运算结果溢出以及操作违反权限等情况。遇到异常发生的情况时,CPU 将暂时中断当前程序,跳转到异常处理程序。这时,CPU 模式会变更到内核模式。异常处理完成后,原则上将返

回异常中断处,但如果发生致命错误会强制中止执行的程序。异常处理的流程如图 1-95 所示。

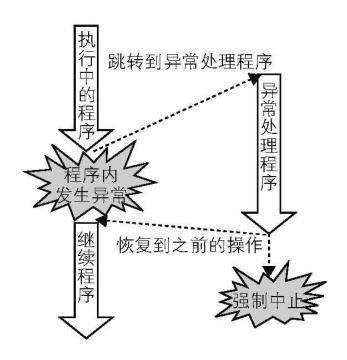


图 1-95 异常处理的流程

中断和异常最大的区别在于发生的原因。中断是由外部因素引起的与正在执行的操作的异步情况,而异常是在正在执行的操作的内部发生的。由于都是暂停正在进行的操作并跳转到处理程序,有着相同的动作特征,中断和异常的处理本质上是一致的。因此,中断和异常使用相同机制不加区分的 CPU 也很多。

• 异常发生时的流水线动作

流水线化的 CPU 在异常发生时的处理稍微有些复杂。异常发生后,导致异常发生的指令以及其后的指令暂停执行,并跳转到异常处理程序。由于流水线化的 CPU 中的后续指令也在执行中,需要先将流水线内所有数据缓存后,再将 PC 寄存器设置为异常处理

程序地址,最后重新启动流水线。有异常发生时的流水线动作如图 1-96 所示。

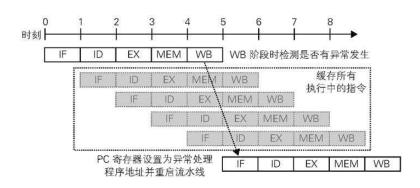


图 1-96 异常发生时的流水线动作

根据异常种类的不同,发生异常的流水线级也不同。因此异常发生时的动作较为复杂。最简便的方式是在流水线寄存器设置专用寄存器以标示异常发生的位置,最后在 WB 阶段检查是否有异常发生。因为操作结果的写回是在 WB 阶段,如果在 WB 级执行前将其内容缓存,指令就可以和从未执行过一样。

但是,也有一个例外。只有写入内存的存储 指令,在 MEM 阶段就会将结果写入内存。 因此,为了使存储指令无效,需要判断内存 写入前的指令是否发生异常。

专栏

CPI 和 MIPS 值

为了表示 CPU 运行一条指令所需的时钟周期,有一个称为 CPI(Clock cycle Per Instruction)的指标。CPI 表示平均一条指令所需的时钟周期,知道了程序行数和 CPI,即可计算出程序执行所需要的时钟周期数。

1.8.1 节介绍的 CPU, 一条指令的执行 需要 5 个时钟周期。由于使用了流水线 技术, 看起来可以同时执行 5 条指令。但是, 由于延迟或缓存会引发流水线冒泡, 实际程序的不同, CPI 会有所变化。

MIPS(Million Instructions Per Second) 是衡量 CPU 性能的指标。MIPS 是表示 每秒可以执行几百万条指令的数值,是 用 CPU 的动作频率除以平均 CPI 计算 得到的。

1.8.2 AZ Processor 的设计

• AZ Processor 的流水线结构

本章基于 RISC 架构的 32 位 CPU,使用 1.8.1 节讲解的典型的 5 级流水线技术制作 AZ Processor。AZ Processor的框图如图 1-97 所示。

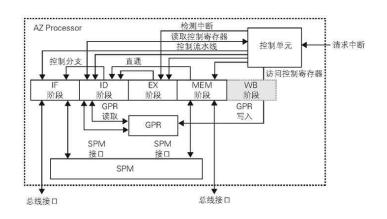


图 1-97 AZ Processor 框图

AZ Processor 由以下部分组成:流水线中的 IF 阶段、ID 阶段、EX 阶段、MEM 阶段、CPU 中的存储器通用寄存器、控制 CPU 的 CPU 控制单元,以及

CPU 可以直接访问的专用存储器 SPM(Scratch Pad Memory)。虚线中的 WB 阶段,实际上在结果写回的通用 寄存器或 CPU 控制单元中实现,这个 模块本身并不存在。

IF 阶段和 MEM 阶段通过总线与内存和 I/O 相连。为了使流水线高效工作,需要每个周期都向流水线提供指令或数据。因此,我们为 AZ Processor 设置可以高速访问的 CPU 专用 SPM。虽然 SPM 和其他内存、I/O 同样分配有地址空间,但 CPU 可以直接访问而不用通过总线。SPM 也有点像缓存,但却是本身分配有地址空间的存储器。

分支的判定在 ID 阶段进行。我们采用了延迟分支机制,也就是说,分支指令的下一条指令被作为延迟间隙执行,以此规避控制冒险。EX 阶段和 MEM 阶段的处理结果可以直通到 ID 阶段,以此规避数据冒险。

流水线寄存器的停滞与刷新和流水线的控制等操作由控制单元负责。控制单元还可以接受来自外部的中断请求,并根据 CPU 的设置输出中断检测信号。中断的检测是在 EX 阶段进行的。

AZ Processor 流水线的细节如图 1-98 所示。

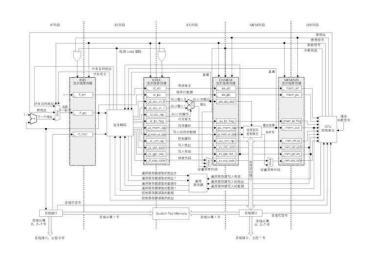


图 1-98 AZ Processor 流水线构造

- AZ Processor 的指令集架构
 - 指令格式一览

AZ Processor 的指令,根据指令二进制代码内信息格式的不同分为5类。指令的格式如图 1-99 所示,指令代码中各字段的说明请参见表1-27。指令的最高 6 位用来定义操作码(operation code),指示指令进行的操作。剩余的位称为操作数(operand),用来表示指令使用的寄存器的地址和立即数等。



图 1-99 指令的格式

表 1-27 指令字段

字段名	位置	位宽	含义
OP (Opecode)	31:26	6	操作码
Ra (Register A)	25:21	5	寄存器 A 的 地址
Rb (Register B)	20:16	5	寄存器 B 的 地址
Rc (Register C)	15:11	5	寄存器 C 的 地址
Immediate	15:0	16	立即数

AZ Processor 指令格式最大可以有3个操作数。指令根据操作数的不同,可以分为5类: R3(3 Registers) 格式、R2(2 Registers) 格式、R1(1 Register) 格式、R0(0 Register) 格式和 R2I(2 Registers & Immediate) 格式。Reserved 区域为保留(未使用)字段。

立即数是指嵌入到指令字段中的常数。程序中经常出现使用常数的运算。例如循环的递增、变量的初始化等众多场合。如果 CPU 的指令只能使用寄存器,则需要将常数存储在内存并在每次使用时加载。这种做法复杂且效率低下,因此指令中嵌入立即数这种做法非常有效。

AZ Processor 指令一览如表 1-28 所示。AZ Processor 有 7 种类型的指令:逻辑运算指令、算术运算指令、移位指令、分支指令、内存访

问指令、特殊指令,以及特权指令。

表 1-28 指令一览

类别	指令	名称	操作码	格式	含义
	ANDR	AND Register	000000 (0x00)	R3	寄存器间的逻辑与
	ANDI	AND Immediate	000001 (0x01)	R2I	寄存器与常数的逻辑与
逻	ORR	OR Register	000010 (0x02)	R3	寄存器间的逻辑或
辑运算指令	ORI	OR Immediate	000011 (0x03)	R2I	寄存器与常数的逻辑或
					寄存器

	XORR	XOR Register	000100 (0x04)	R3	间的逻辑异或
	XORI	XOR Immediate	000101 (0x05)	R2I	寄存器与常数的逻辑异或
	ADDSR	Add Signed Register	000110 (0x06)	R3	寄存器间的有符号加法
	ADDSI	Add Signed Immediate	000111 (0x07)	R2I	寄存器与常数的有符号加法
算术	ADDUR	Add Unsigned Register	001000 (0x08)	R3	寄存器间的无符号加

运					法
算指令	ADDUI	Add Unsigned Immediate	001001 (0x09)	R2I	寄存器与常数的无符号加法
	SUBSR	Subtract Signed Register	001010 (0x0A)	R3	寄存器间的有符号加法
	SUBUR	Subtract Unsigned Register	001011 (0x0B)	R3	寄存器间的无符号加法
	SHRLR	SHift Right Logical Register	001100 (0x0C)	R3	寄存器间的逻辑右移
	SHRLI	SHift Right	001101	R2I	寄存器与常数

移位指		Logical Immediate	(0x0D)		的逻辑右移
令	SHLLR	SHift Left Logical Register	001110 (0x0E)	R3	寄存器间的逻辑左移
	SHLLI	SHift Left Logical Immediate	001111 (0x0F)	R2I	寄存器与常数的逻辑左移
	BE	Branch Equal	010000 (0x10)	R2I	根据寄存器间的比较决定分支
	BNE	Branch Not Equal	010001 (0x11)	R2I	根据寄存器间的比较决

					定分支
分支指令	BSGT	Branch Signed Greater Than	010010 (0x12)	R2I	根据寄存器间的有符号比较决定分支
	BUGT	Branch Unsigned Greater Than	010011 (0x13)	R2I	根据寄存器间的无符号比较决定分支
分	JMP	JuMP	010100 (0x14)	R1	跳转到寄存器指定的地址
分支指令					调用

	CALL	CALL	010101 (0x15)	R1	寄存器指定地址的子程序
内存访	LDW	LoaD Word	010110 (0x16)	R2I	字读取
问指令	STW	STore Word	010111 (0x17)	R2I	字写入
特殊指令	TRAP	TRAP	011000 (0x18)	R0	陷阱指令
	RDCR	ReaD Control Register	011001 (0x19)	R2	读取控制寄存器
特权指令	WRCR	WRite Control Register	011010 (0x1A)	R2	写入控制寄存器
	EXRT	EXception ReTurn	011011 (0x1B)	R0	从异常恢复

• 逻辑运算指令

逻辑运算指令对作为操作数的寄存器之间,或者寄存器与立即数之间

进行逻辑运算,并将结果存入寄存器。逻辑运算指令有针对寄存器间逻辑运算的 R3型,也有针对寄存器与立即数间逻辑运算的 R2I型。表 1-29 列出了逻辑运算指令。

表 1-29 逻辑运算指令

	31	26 25	21	20	16 15	1	1 10	0
ANIDO (ANID SELESO)	000000(0x0	0)	Ra	Rb		Rc	000_0000_000	00
ANDR (AND 寄存器)	ANDR GPR[Ra] 和 G						保留	
	31	26 25	21	20	16 15			0
ANDI (AND 立即数)	000001(0x0	1)	Ra	Rb		XXXX_)	XXXX_XXXX	
ANDI (AND MASK)	ANDI GPR[Ra] 和 0	寄- 扩充后的	存器 A 立即数逻	寄存器 辑与,结果	B 表写入 0	SPR[Rb]。	立即数	
	31	26 25	21	20	16 15	1	1 10	0
ODD (OD th # 188)	000010(0x0	2)	Ra	Rb		Rc	000_0000_000	00
ORR (OR 寄存器)	ORR GPR[Ra] 和 G						保留	
	31	26 25	21	20	16 15			C
ORI (OR 立即数)	000011(0x0	3)	Ra	Rb		XXXX_	XXXX_XXXX_XXXX	
ORI (OR 立即数)	ORI GPR[Ra] 和 0			寄存器 辑或,结别			立即数	
	31	26 25	21	20	16 15	1	1 10	C
VODD WOD thit m	000100(0x0	4)	Ra	Rb		Rc	000_0000_000	00
XORR (XOR 寄存器)	XORR GPR[Ra] 和 G			寄存器! 结果写入:			保留	
	31	26 25	21	20	16 15			C
XORI (XOR 立即数)	000101(0x0	5)	Ra	Rb		XXXX_	XXXX_XXXX	
AUNI (AUN 址即数)	XORI GPR[Ra] 和 0			寄存器! 辑异或, 组			立即数。	

含有立即数的指令需要将 16 位的立即数扩充到 32 位后参与运算。扩充的方法有两种,一种是高 16 位全部用 0 填充的 0 扩充,一种是高 16 位用 MSB(符号位)填充的符号扩充。0 扩充和符号扩充的示意图请参见图 1-100。逻辑运算指令中对立即数使用 0 扩充。

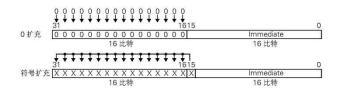


图 1-100 0 扩充和符号扩充

• 算术运算指令

算术运算指令对作为操作数的寄存器之间,或者寄存器与立即数之间

进行算术运算,并将结果存入寄存器。算术运算指令有针对寄存器间算术运算的 R3型,也有针对寄存器与立即数间算术运算的 R2I型。表 1-30 列出了算术运算指令。

表 1-30 算术运算指令

	31	2625	2	120	1615	1110)	0
ADDSR	000110 (0x06)	Ra	Rb	F	Rc	000_0000_000	00
(ADD Signed 寄存器)	GPR[Ra]	SR 寄 ラ GPR[Rb] 益出,产生》	相加,结	果写入 GP		器 C	保留	
	31	2625	2	120	1615			0
ADDSI	000111 (0x07)	Ra	Rb	X	XXX_XXX	X_XXXX_XXXX	
(ADD Signed 立即数)	GPR[Ra]	SI 寄 与符号扩充/ 益出,产生/	舌的 立即	数相加,结	B 果写入 GF	立 PR[Rb]。	即数	
		2625						0
ADDUR							000_0000_000	00
(ADD Unsigned 寄存器)		JR 為 写 GPR[Rb]				器 C	保留	
	31	2625	2	120	1615			0
ADDUI	001001 (0x09)	Ra	Rb	X	XXX_XXX	X_XXXX_XXXX	
(ADD Unsigned 立即数)		UI 寄 与符号扩充/					即数	,,
	31	2625	2	120	1615	11 10)	0
SUBSR	001010 (0x0A)	Ra	Rb	F	Rc	000_0000_000	00
(SUBtract Signed 寄存器)	GPR[Ra]	SR 寄 写 GPR[Rb] 益出,产生》	相加,结	果写入 GP		器 C	保留	
	31	2625	2	120	1615	1110)	0
SUBUR	001011 (0x0B)	Ra	Rb	F	Rc	000_0000_000	00
(SUBtract Unsigned 寄存器)		JR 寄 写 GPR[Rb]				器 C	保留	

加法指令和减法指令分为有符号与无符号两类。这两种指令的区别在于是否检测溢出。溢出是指运算结果超出寄存器或内存可以表示的范围。

下面以 8 位数据间的加法运算为例进行说明。Verilog HDL 中以8'b01100100的形式描述常数。例如,100(8'b01100100)加64(8'b010000000)结果为164(8'b10100100)。观察结果的二进制序列 8'b10100100可以发现,发生了向 MSB(符号位)的进位。补码的 8'b10100100十进制值为 -92,不是正确答案。因为有符号 8 位整数的表现范围为-128~127,正确答案 164 不在此范围内。

加法运算发生溢出有两种情况,"正数加正数得到负数"或"负数加负数得到正数"。减法运算发生溢出的情况有"负数减正数得到正数"或"正数减负数得到负数"。也就是说,如果运算结果的符号发生错误就会产生溢出。有符号指令需要检测溢出。如果运算结果有溢出,则产生溢出异常。

寄存器与立即数间的算术运算指令,立即数采用符号扩充。因此寄存器与立即数的算术运算指令中没有减法指令。立即数与负数相加和减法运算是等效的。

• 移位指令

移位指令对作为操作数的寄存器之间,或者寄存器与立即数之间进行移位,并将结果存入寄存器。移位是将二进制序列整体向左或向右移动的操作。序列向左移动称为左移,向右移动称为右移。图 1-101为移位的示例。移出的比特被废弃,移动产生的空位重新插入 0 或 1。空位插入 0 的移位称为逻辑移位。

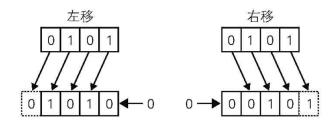


图 1-101 移位操作

AZ Processor 的移位指令有针对寄

存器间移位的 R3 型,也有针对寄存器与立即数间移位的 R2I 型。表1-31 列出了移位指令。32 位的二进制序列最大可以移动 32 位。因此位移量用寄存器或立即数的最低5位(2的5次方为32)表示。

表 1-31 移位指令

	31 26	25 21	20 16	15 11	10 0
SHRLR	001100 (0x0C)	Ra	Rb	Rc	000_0000_0000
(SHift Right Logical 寄存器)	SHRLR 对 GPR[Ra] 右移,	寄存器 A 结果写入 GP			<i>保留</i> 低 5 位指定。
	31 26	25 2	20 16	15	0
SHRLI	001101 (0x0D)	Ra	Rb	XXXX_XX	XXX_XXXX_XXXX
SHift Right Logical 立即数)	SHRLI 对 GPR[Ra] 右移,	寄存器 A 结果写入 GP	寄存器 B R[Rb]。位移量	由立即数的低	立即数 5 位指定。
	31 26	25 2	20 16	15 11	10 0
SHLLR	001110 (0x0E)	Ra	Rb	Rc	000_0000_0000
(SHift Left Logical 寄存器)	SHLLR 对 GPR[Ra] 左移,		寄存器 B R[Rc]。位移量		<i>保留</i> 低 5 位指定。
	31 26	25 2	20 16	15	0
SHLLI	001111 (0x0F)	Ra	Rb	XXXX_XXX	XX_XXXX_XXXX
(SHift Left Logical 立即数)	SHLLI 对 GPR[Ra] 左移。	寄存器 A 结果写入 GP			文即数 5 位指定。

• 分支指令

分支指令是改变程序流程的指令。如果分支成立,那么下一条将要执行的指令就会被改变。因为 AZ Processor 采用了延迟分支处理,如果分支成立,要等到分支指令的下一条指令执行后再跳转到分支指令的指令。分支指令有 R2I 型条件分支指令和 R1 型无条件分支指令两种。分支指令如表 1-32 所示。

表 1-32 分支指令

	31	26 25	21	20	16 15		0
05 /0 / 5 /	010000 (0)	(10)	Ra	R	b	XXXX_XXXX_XXXX_XXXX	
BE (Branch Equal)	BE GPR[Ra] 等于		寄存器 A (b) 时,跳车			立即数	
	31	26 25	5 21	20	16 15		0
	010001 (0)	(11)	Ra	Ri	b	XXXX_XXXX_XXXXX_XXXX	
BNE (Branch Not Equal)	BNE GPR[Ra] 与 (寄存器 A] 不相等时,			立即数	
	31	26 25	21	20	16 15		0
BSGT	010010 (0)	(12.)	Ra	RI	b	XXXX_XXXXX_XXXXX_XXXX	
(Branch Signed Greater Than)	BSGT GPR[Rb] 比(进行有符号)	GPR[Ra	寄存器 A] 大时,跳等			立即数	
	31				16 15		0
BUGT	010011 (0)	(13)	Ra	RI	b	XXXX_XXXX_XXXX_XXXX	
			寄存器A	寄存		立即数	
(Branch Unsigned Greater Than)	GPR[Rb] 比 (进行无符号)	GPR[Ra]	大时,跳车	封目标	地址。		
Branch Unsigned Greater Than)	GPR[Rb] 比 (进行无符号) 31	GPR[Ra] 比较。 26 25	5 21		地址。		0
200 2000 TATE FAMILIE	GPR[Rb] 比 (进行无符号) 31 010100 (0)	GPR[Ra] 比较。 26 25 ×14)	5 21 Ra			0000_0000_0000_0000	0
(Branch Unsigned Greater Than) JMP (JuMP)	GPR[Rb] 比 (进行无符号) 31	GPR[Ra] 比较。 26 25 ×14)	5 21 Ra 寄存器 A	20		0000_0000_0000 保留	0
200 2000 TATE FAMILIE	GPR[Rb] 比 (进行无符号) 31 010100 (0) JMP 无条件跳转至	GPR[Ra] 比较。 26 25 ×14)	5 21 Ra 寄存器 A	也址。			0
200 2000 TATE FAMILIE	GPR[Rb] 比 (进行无符号) 31 010100 (0) JMP 无条件跳转至	SPR[Ra] 比较。 26 25 (14)	i 21 Ra 寄存器 A Ra] 指定的均	120 世址。 120	0_0000_		

条件指令对寄存器进行比较,如果 条件成立则跳转到目标地址。目标 地址由 PC 寄存器与符号扩充后的 立即数相加得到。立即数字段中指 定的地址基于字(32 位)编址方 式进行计算,每个字分配一个地 址。

目标地址要利用流水线寄存器中PC的值进行计算。因为PC中存放的是下一条指令的地址,所以目标地址为"下一条指令的地址+立即数"。使用PC值分支跳转到相对位置的方法称为PC相对分支。

BE 指令在寄存器间的值相等和BNE 指令在寄存器间的值不等时分支成立。BSGT 指令与 BUGT 指令对通用寄存器(GPR)间的值进行比较,当条件 GPR[Ra] <GPR[Rb] 成立时分支成立。只是BSGT 指令将寄存器的值作为有符号数值进行比较,而 BUGT 指令将寄存器的值作为无符号数值进行比较。

无条件分支指令会强制跳转程序。

分支目标地址在寄存器中指定,这种分支称为寄存器间接分支。JMP指令用来强制跳转到寄存器指定的地址。CALL指令用来调用寄存器指定地址处的子程序。子程序的调用是指先执行子程序,处理完成后返回到调用处的操作。

JMP 指令与 CALL 指令都是无条件跳转语句,在这一点上两者效果是相同的。不同之处在于 CALL 指令在 GPR31 寄存器中存放两条之后指令的地址。由于 CALL 的下一条指令会被当作延迟间隙执行,所以 GPR31 中存放的地址为"CALL 指令地址 +8"。因为存放了子程序执行完成后返回。在返回时,使用通用寄存器 31 作为操作数并执行 JMP 指令。图 1-102 为子程序调用流程。

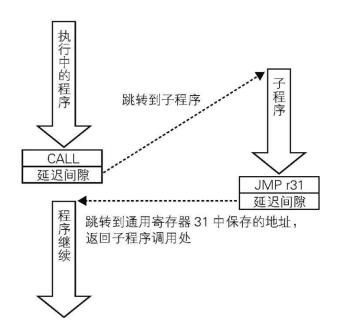


图 1-102 子程序调用流程

• 内存访问指令

内存访问指令用来从内存读取数据或向内存写入数据。内存访问指令格式为 R2I 型。表 1-33 列出了内存访问指令。

表 1-33 内存访问指令

	31 26	25 21	20 1618	5 0
	010110 (0x16)	Ra	Rb	XXXX_XXXXX_XXXXX_XXXX
	LDW	寄存器 A	寄存器 B	立即数
LDW (LoaD Word)	GPR[Ra] 和符号却 从地址指定的内容 如果地址没有按	字中读取 1 个字字边界对齐,产	的数据并存入 GI 生未对齐异常。	323334
		25 21		
	010111 (0x17) Ra	Rb	XXXX_XXXX_XXXX_XXXX
STW (STore Word)	STW	寄存器 A	寄存器 B	立即数
	GPR[Ra] 和符号加 向地址指定的内存 如果地址没有按	字中写入 GPR[R	bl 中的一个字的	数据

LDW 指令用来从内存中读取 1 个字的数据并存入寄存器中,读取地址由寄存器与符号扩充后的立即数相加得到。STW 指令用来将寄存器中 1 个字的数值写入内存中,写入地址由寄存器与符号扩充后的立即数相加得到。这种地址指定的方式称为有偏移量的寄存器间接寻址。

执行内存访问指令时要对地址进行 对齐检测。如果访问未对齐的地 址,则会产生未对齐异常。对齐是 指要访问数据的位置在单位数据的 边界上。如果访问的地址跨过单位 数据的边界线则称为未对齐。

图 1-103 为对齐的示例。在按照字节对齐的地址空间中访问 1 个字(4 字节)的数据时,如果起始地址为 0x00,所访问的数据位于0x00 到 0x03。这时数据起始于字的边界,是对齐的。字边界是从 0 开始 1 个字长的区间。假如从

0x01 开始访问 1 个字的数据,所访问的数据位于 0x01 到 0x04。这时,由于 0x04 属于下一个字的空间,数据跨越了字的边界。这种访问的地址就是未对齐的。同样,从 0x02 开始的 1 个字都是未对齐的。

未对齐会引起多次内存访问的问题。比如要访问从 0x01 开始访问 1 个字的数据,而 0x01 至 0x03 与 0x04 存放在不同的内存地址中。因此需要访问内存两次然后将数据进行组合。如果允许这种操作,硬件设计会变得复杂。因此在内存访问指令中进行对齐的检查,如果发生访问未对齐地址的情况,则产生未对齐异常。

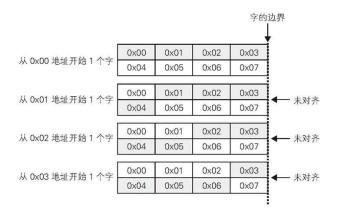


图 1-103 对齐

• 特殊指令

特殊指令是用来故意引发异常的指令,它的主要用途是变更 CPU 模式。故意引发异常会转移到内核模式。AZ Processor 支持的特殊指令是称为 TRAP 的指令,执行 TRAP

指令的话会引发陷阱异常。由于TRAP指令只用来引发异常,属于没有操作数的RO型指令。系统调用指令如表 1-34 所示。

表 1-34 特殊命令

	31 26 25		0
TRAP (TRAP)	011000 (0x18)	00_0000_0000_0000_0000_0000_0000	
THAT (THAT)	TRAP 引发陷阱异常。	保留	

• 特权指令

特权指令是只能在内核模式执行的特殊指令。通过特权指令可以实现 CPU 控制寄存器访问、从异常恢复等控制 CPU 状态的操作。特权指令如表 1-35 所示。

RDCR 指令用来读取控制寄存器的值并写入通用寄存器; WRCR 指令用来将通用寄存器的值写入控制寄存器; EXRT 指令用来从异常恢复。由于特权指令只能在内核模式执行,如果在用户模式执行会引发特权违反异常。

表 1-35 特权指令

	31 26	25 21	20 16	15	0
RDCR	011001 (0x19)	Ra	Rb	0000_0000_0000_0000	
(ReaD Control 寄存器)	RDCR 将 CTRL[Ra] 的数据	寄存器 A 居写入 GPR[Ri	寄存器 B bl。	保留	
	31 26	25 21	20 16	15	0
WRCR	011010 (0x1a)	Ra	Rb	0000_0000_0000_0000	
(WRite Control 寄存器)	WRCR 将 GPR[Ra] 的数据	寄存器 A 写入 CTRL[R	寄存器 B bl。	保留	
9001000000	31 26	25			0
EXRT	011011 (0x1b)	00	0_0000_0000_0	0000_0000_0000_0000	
(EXception ReTurn)	EXRT 从异常恢复。			保留	

• 异常

AZ Processor 的异常一览如表 1-36 所示。AZ Processor 的中断也与异常一样处理。CPU 中发生异常时,

先将异常发生处指令的地址写入 PC 寄存器,再将 CPU 模式变更到 内核模式,最后跳转到异常向量的 地址。异常向量是指异常处理程序 的起始地址。

表 1-36 异常一览

异常	说明	可能引发 该异常的 指令	异常 代码
无异常	没有异常发 生的状态	-	0x0
外部中 断	发生外部中 断时发生	-	0x1
未定义 指令	解码未定义 指令时发生	-	0x2
算术溢出	发生算术溢 出时	ADDSR, ADDSI, SUBSR	0x3
地址未 对齐	访问未对齐 地址时发生	LDW, STW	0x4
陷阱	执行 TRAP 指令时	TRAP	0x5
特权违反	在 User Mode 执行特权指 令时发生	RDCR, WRCR, EXRT	0x6

专栏

指令集架构与微架构

CPU 架构(Architecture)大概 分为指令集架构(Instruction Set Architecture)与微架构 (Micro Architecture)两种。 指令集架构是从 CPU 所支持 的指令集合、寄存器、异常以 及中断等程序员的角度着眼的

架构。反之,微架构是较指令 集架构更底层,从实际硬件角 度着眼的架构。

1.8.3 AZ Processor 的实现

• CPU 全局使用的宏

CPU 代码全局使用的宏记述在 isa.h 和 cpu.h 两个文件中。 isa.h 中记载的是与指令集架构 有关的宏,cpu.h 中记载的是与微架构有关的宏。表 1-37 与表 1-38 分别列出了 isa.h 与 cpu.h 的内容。

表 1-37 宏一览(cpu.h)

宏名	值
REG_NUM	32
REG_ADDR_W	5
RegAddrBus	4:0
CPU_IRQ_CH	8
ALU_OP_W	4
AluOpBus	3:0
ALU_OP_NOP	4'h0
ALU_OP_AND	4'h1
ALU_OP_OR	4'h2
ALU_OP_XOR	4'h3
ALU_OP_ADDS	4'h4
ALU_OP_ADDU	4'h5

ALU_OP_SUBS	4'h6
ALU_OP_SUBU	4'h7
ALU_OP_SHRL	4'h8
ALU_OP_SHLL	4'h9
MEM_OP_W	2
MemOpBus	1:0
MEM_OP_NOP	2'h0
MEM_OP_LDW	2'h1
MEM_OP_STW	2'h2
CTRL_OP_W	2
CtrlOpBus	1:0
CTRL_OP_NOP	2'h0
CTRL_OP_WRCR	2'h1
CTRL_OP_EXRT	2'h2
CPU_EXE_MODE_W	1
CpuExeModeBus	0:0
CPU_KERNEL_MODE	1'b0
CPU_USER_MODE	1'b1
CREG_ADDR_STATUS	5'h0
CREG_ADDR_PRE_STATUS	5'h1
CREG_ADDR_PC	5'h2
CREG_ADDR_EPC	5'h3
CREG_ADDR_EXP_VECTOR	5'h4

CREG_ADDR_CAUSE	5'h5
CREG_ADDR_INT_MASK	5'h6
CREG_ADDR_IRQ	5'h7
CREG_ADDR_ROM_SIZE	5'h1d
CREG_ADDR_SPM_SIZE	5'h1e
CREG_ADDR_CPU_INFO	5'h1f
CregExeModeLoc	0
CregIntEnableLoc	1
CregExpCodeLoc	2:0
CregDlyFlagLoc	3
BusIfStateBus	1:0
BUS_IF_STATE_IDLE	2'h0
BUS_IF_STATE_REQ	2'h1
BUS_IF_STATE_ACCESS	2'h2
BUS_IF_STATE_ACCESS BUS_IF_STATE_STALL	2'h2 2'h3
BUS_IF_STATE_STALL	2'h3
BUS_IF_STATE_STALL RESET_VECTOR	2'h3 30'h0
BUS_IF_STATE_STALL RESET_VECTOR ShAmountBus	2'h3 30'h0 4:0
BUS_IF_STATE_STALL RESET_VECTOR ShAmountBus ShAmountLoc	2'h3 30'h0 4:0 4:0
BUS_IF_STATE_STALL RESET_VECTOR ShAmountBus ShAmountLoc RELEASE_YEAR	2'h3 30'h0 4:0 4:0 8'd41

表 **1-38** 宏一览(isa.h)

宏名	值	Ę
ISA_NOP	32'h0	NoO
ISA_OP_W	6	操作
IsaOpBus	5:0	操作
IsaOpLoc	31:26	操作
ISA_OP_ANDR	6'h00	寄存逻辑
ISA_OP_ANDI	6'h01	寄存数与
ISA_OP_ORR	6'h02	寄存逻辑
ISA_OP_ORI	6'h03	寄存数间或
ISA_OP_XORR	6'h04	寄存逻辑
ISA_OP_XORI	6'h05	寄有
ISA_OP_ADDSR	6'h06	寄存有符
ISA_OP_ADDSI	6'h07	寄存数间号加
ISA_OP_ADDUR	6'h08	寄存无符
ISA_OP_ADDUI	6'h09	寄存 数间 号加
ISA_OP_SUBSR	6'h0a	寄存有符
ISA_OP_SUBUR	6'h0b	寄存无符
ISA_OP_SHRLR	6'h0c	寄存逻辑
ISA_OP_SHRLI	6'h0d	寄存 数间 右移
ISA_OP_SHLLR	6'h0e	寄存

		逻辑
ISA_OP_SHLLI	6'h0f	寄存 数间 左移
ISA_OP_BE	6'h10	寄存比较
ISA_OP_BNE	6'h11	寄存比较
ISA_OP_BSGT	6'h12	寄存 有符 (<)
ISA_OP_BUGT	6'h13	寄存 无符 (<)
ISA_OP_JMP	6'h14	寄存的绝
ISA_OP_CALL	6'h15	寄存 的子 用
ISA_OP_LDW	6'h16	字读
ISA_OP_STW	6'h17	字写
ISA_OP_TRAP	6'h18	陷阱
ISA_OP_RDCR	6'h19	读取 存器
ISA_OP_WRCR	6'h1a	写入 存器
ISA_OP_EXRT	6'h1b	从异
ISA_REG_ADDR_W	5	寄存宽
IsaRegAddrBus	4:0	寄存总线
IsaRaAddrLoc	25:21	寄存的位
IsaRbAddrLoc	20:16	寄存的位
IsaRcAddrLoc	15:11	寄存的位
ISA_IMM_W	16	立即
ISA_EXT_W	16	符号的立

ISA_IMM_MSB	15	立即位
IsaImmBus	15:0	立即
IsaImmLoc	15:0	立即
ISA_EXP_W	3	异常
IsaExpBus	2:0	异常 线
ISA_EXP_NO_EXP	3'h0	无异
ISA_EXP_EXT_INT	3'h1	外部
ISA_EXP_UNDEF_INSN	3'h2	未定
ISA_EXP_OVERFLOW	3'h3	溢出
ISA_EXP_MISS_ALIGN	3'h4	地址
ISA_EXP_TRAP	3'h5	陷阱
ISA_EXP_PRV_VIO	3'h6	违反

• 通用寄存器

我们首先制作作为 CPU 存储 区域的通用寄存器。AZ Processor 的指令最大可以指定 三个寄存器作为操作数,从指定 三个寄存器读取值,然后的 另一个寄存器写入值。因此写 存器堆需要有两个读取端口的 存器堆需要有两个读取品的 信号线一览如表 1-39 所示, 源程序如代码 1-11 所示。

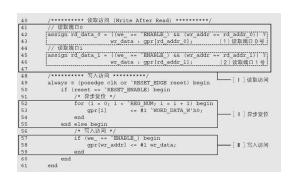
表 1-39 信号一览 (gpr.v)

组	信号名	信号类型	数据类型	位宽	含义
时钟	clk	输入端口	wire	1	时钟

H					
与复位	reset	输入端口	wire	1	异步复位
读取端	rd_addr_0	输入端口	wire	5	读取的地址
口 0	rd_data_0	输出端口	wire	32	读取的数据
读取端	rd_addr_1	输入端口	wire	5	读取的地址
1	rd_data_1	输出端口	wire	32	读取的数据
	we_	输入端口	wire	1	写入有效信号
写入端口	wr_addr	输入端口	wire	32	写入的地址
	wr_data	输入端口	wire	32	写入的数据
内	gpr	内部信号	reg	32x32	寄存器序列



代码 **1-11** 通用寄存器 (**gpr.v**)



[I] 读取访问

- (1) 处对读取端口 0 号、
- (2)处对读取端口1号进行读取访问。如果在读取的同时对相同地址进行写入操作,则直接将写入的数据输出。当写入有效信号(we_)有效,并且写入地址(wr_addr)和读取地址(rd_addr_0或rd_addr_1)一致时,写入的数据(wr_data)输出到输出数据(rd_data_0或rd_data_1)。

[II] 异步复位

全部寄存器的值初始化为 0。 使用 for 语句遍历所有寄存器 进行初始化操作。

[III] 写入访问

当写入有效信号(we_)有效时,向指定的写入地址(wr_addr)写入数据(wr_data)。

• SPM

SPM(Scratch Pad Memory) 是 CPU 可以不经过总线直接 访问的专用内存。SPM 使用 一个名为 spm 的模块构成。存 储器使用 FPGA 的 Dual Port RAM 实现。表 1-40 为 spm 使 用的宏一览,表 1-41 为信号 一览,代码 1-12 为源程序。

表 1-40 宏一览 (spm.h)

宏名	值	含义
SPM_SIZE	16384	SPM 的容量
SPM_DEPTH	4096	SPM 的深度
SPM_ADDR_W	12	地址宽
SpmAddrBus	11:0	地址总 线
SpmAddrLoc	11:0	地址的 位置

表 **1-41** 信号线一览 (**spm.v**)

组	信号名	信号类型	数据类型	在货

时钟	clk	输入端口	wire	1
	if_spm_addr	输入端口	wire	1.
	if_spm_as_	输入端口	wire	1
A端口IF阶	if_spm_rw	输入端口	wire	1
段	if_spm_wr_data	输入端口	wire	3.
	if_spm_rd_data	输出端口	wire	3.
	mem_spm_addr	输入端口	wire	1
	mem_spm_as_	输入端口	wire	1
B端 口 MEM 阶段	mem_spm_rw	输入端口	wire	1
	mem_spm_wr_data	输入端口	wire	3.
		输出		

	mem_spm_rd_data	端口	wire	3.
A端 口	wea	内部信号	reg	1
B端 口	web	内部信号	reg	1

代码 1-12 Scratch Pad Memory (spm.v)

[I] A 端口写入有效信号的生成

当来自 IF 阶段的地址有效信号(if_spm_as_)有效、读/写信号(if_spm_rw)为写入(WRITE)时,写入有效信号(wea)为有效。基本上 IF 阶段只进行指令的读取,因此只有存储器读取操作。

[II]**B**端口写入有效信号的 生成

当来自 MEM 阶段的地址有效信号(mem_spm_as_)有效、读/写信号(mem_spm_rw)为写入(WRITE)时,写入有效信号(web)为有效。

[III] 存储器的实例化

实例化赛灵思 FPGA 的块RAM 模块。

• 总线接口

总线接口用来对总线的访问进行控制。CPU在IF阶段和MEM阶段访问内存。总线接口接受来自CPU的内存访问请求,并控制其对总线的访问。

因为 AZ Processor 内置了 SPM,总线接口要根据访问的 地址选择总线和 SPM 的访 问。因为 CPU 与 SPM 直接连 接,CPU 对 SPM 进行读写只 需要一个周期。访问总线时需 要遵循总线协议进行访问控 制。

在总线空闲状态的前提下,当 未在执行刷新流水线操作、地 址选通有效以及对1号之外的 总线从属进行访问时,可以进 行总线访问。当正在执行刷新 行总线访问。CPU 要访问总线 按性行访问。CPU 要访问总线 时总线接口转移到总线请求。 对总线控制权进行请求。

如果总线许可信号有效,则表明总线控制权申请成功,总线接口转移到总线访问状态进行总线访问。最后,总线访问完成后使能就绪信号。这时完成后使能就结信号。这时完如果流水线在延迟状态等待延迟状态等待到延迟状态等待延迟,则是不够。如果未发生延迟,则返回空闲状态。

总线接口的状态迁移图如图 1-104 所示,信号一览如表 1-42 所示。

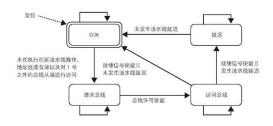


图 **1-104** 总线接口的状态迁 移图

表 1-42 信号线一览 (bus_if.v)

分组	信号名	信号类型	数据类型	位宽	含义
时钟复位	clk	输入端口	wire	1	时钟
	reset	输入端口	wire	1	异步 复位
		输			

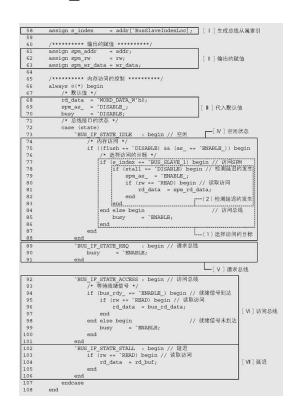
	stall	入端口	wire	1	延迟信号
流线信号	flush	输入端口	wire	1	刷新信号
	busy	输出端口	reg	1	总线 忙信 号
	addr	输入端口	wire	30	CPU : 地 址
	as_	输入端口	wire	1	CPU : 地 址有 效
CPU 接口	rw	输入端口	wire	1	CPU :读 /写
	wr_data	输入端口	wire	32	CPU : 入 数据
	rd_data	输出端口	reg	32	CPU : 取 数据
	spm_rd_data	输入端口	wire	32	SPM : 读 取的 数据
	spm_addr	输出端口	wire	30	SPM :地 址
SPM 接口	spm_as_	输出端口	reg	1	SPM : 地 址选 通
		输出			SPM

	spm_rw	端口	wire	1	: 读 /写
	spm_wr_data	输出端口	wire	32	SPM :写 入的 数据
	bus_rd_data	输入端口	wire	32	总线 : 取数据
	bus_rdy_	输入端口	wire	1	总线 : 就 绪
	bus_grnt_	输入端口	wire	1	总线 : 可
	bus_req_	输出端口	reg	1	总线 : 请 求
总线 接口	bus_addr	输出端口	wire	30	总线 : 地 址
	bus_as_	输出端口	reg	1	总线: 址选通
	bus_rw	输出端口	wire	1	总线 : 读 /写
	bus_wr_data	输出端口	wire	32	总:入数
	state	内部信号	reg	2	总线 接口 状态
	rd_buf	内部	reg	32	读取

内部信号		信号			缓冲
百 夕	s_index	内部信号	wire	3	总线 从属 索引

总线接口由两部分组成,一部分是控制内存访问的组合电路,另一部分是控制总线接口状态的时序电路。内存访问控制部分的程序如代码 1-13 所示。

代码 **1-13** 内存访问控制 (**bus_if.v**)



[I] 生成总线从属索引

使用 PC 寄存器最高 3 位生成总线从属索引。

[II] 输出的赋值

将输入的地址(addr)、读/写(rw)和写入的数据(wr_data)信号输出到SPM。

[III] 代入默认值

读取的数据(rd_data)初始化为 0, SPM 的地址选通信号(spm_as_)和总线忙信号(busy)设置为无效。

[IV] 空闲状态

空闲状态下,如果刷新信号 (flush)无效且地址选通信号 (as_)有效时,发生内存访 问操作。

- (1)处对即将访问的总线从 属进行选择。当选中 1 号总线 从属时为访问 SPM。SPM 需 要在流水线非延迟的状态下的 高中延迟状态中的流水线 寄存器无法更新,如果这时允 寄存器无法更新,如果这时允 问同一地址。因此 CPU 需要 等待延迟状态解除,在流水线 寄存器可以更新时访问总线。
- (2)处对是否有延迟的发生进行检测。如果是读取访问,则将从 SPM 读取的数据(spm_rd_data)输出到数据输出端口(rd_data)。由于SPM 访问在一个周期即可完

成,不需要使能总线忙信号 (busy)。如果不是访问1号 总线从属,则需要访问总线, 并使能总线忙信号(busy)。

[V] 请求总线

总线访问正在进行时,总线忙信号(busy)有效。

[VI] 访问总线

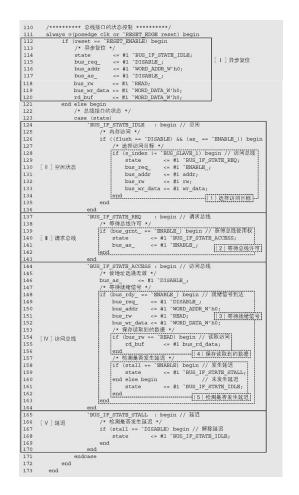
就绪信号(bus_rdy_)使能时,总线访问结束。读/写信号(rw)为读取(READ)时,总线上的读取数据(bus_rd_data)的值输出到读取端口(rd_data)。就绪信号(bus_rdy_)无效时,说明总线访问正在进行,使能总线忙信号(busy)。

[W]] 延迟

在等待延迟解除时,如果读/写信号(rw)为读取 (READ),因为总线访问已 经结束,直接将缓冲 (rd_buf)中的数据输出到读 取端口(rd_data),并使总线 忙信号无效。

总线接口控制部分的程序如代码 1-14 所示。

代码 **1-14** 总线接口控制 (**bus_if.v**)



[] 异步复位

复位信号(reset)有效时,寄存器将被初始化。该初始化操作会将总线接口状态(state)设置为空闲状态

(BUS_IF_STATE_IDLE), 将总线请求信号(bus_req_) 与地址选通信号(bus_as_) 设置为无效,读/写信号 (bus_rw)设置为读取 (READ),将地址 (bus_addr)、写入的数据 (bus_wr_data)、读取缓冲 (rd_buf)清空为0。

[II] 空闲状态

在空闲状态下,如果刷新信号 (flush)无效、地址选通信号 有效,则会发生内存访问操 作。(1)处选择要访问的总 线从属。当访问目标是1号之 外的总线从属时,则会访问总 线。访问总线时使能总线请求 信号(bus_req_),状态转移 到总线请求

(BUS_IF_STATE_REQ)状态。同时,将 CPU 的输出代入地址信号(bus_addr)、读写信号(bus_rw)和写入数据信号(bus_wr_data)。

[III] 请求总线

(2)处如果总线许可(bus_grnt_)有效,状态则会转移到总线访问状态(BUS_IF_STATE_ACCESS)且总线地址选通信号转为(bus as)有效。

「Ⅳ] 访问总线

接下来将总线地址选通信号(bus_as_)设为无效,在(3)处等待就绪信号(bus_rdy_)。一旦就绪信号(bus_rdy_)有效,总线请求信号(bus_req_)则会无效,并释放总线。然后对地址(bus_addr)、读写信号(bus_wr_data)初始化。如果是读取访问的话,在(4)处将读取的数据(bus rd data)

保存到读取缓存(rd_buf)中。

总线访问完成时,如果流水线处于延迟状态,则等待延迟的解除。这样是为了避免延迟中对同一地址反复访问。(5)处对是否发生延迟进行检测。延迟信号(stall)有效时,状态迁移到延迟状态(BUS_IF_STATE_STALL);如果延迟信号(stall)转为无

如果延迟信号(stall)转为无效,则状态转移到空闲状态 (BUS_IF_STATE_IDLE)。

[V] 延迟

等待延迟状态的解除。如果延迟信号(stall)转为无效,则状态转移到空闲状态 (BUS_IF_STATE_IDLE)。

• Instruction Fetch (IF) 阶段

IF 阶段的操作有取指令,并决定下一条 PC 寄存器的内容。 IF 阶段由流水线寄存器与总线接口组成。表 1-43 列出了 IF 阶段的模块一览。

表 1-43 IF 阶段模块一览

模块名	文件名	说明
if_stage	if_stage.v	IF 阶段顶层 模块
if_reg	if_reg.v	IF 阶段流水 线寄存器
bus_if	bus_if.v	总线接口

IF 阶段是根据 PC 寄存器的值进行指令读取的。因为要先确定 PC 的值才可以进行指令读取,因此,指令存储到指令寄存器中的操作发生在 PC 值确定后的下一个时钟周期。这样,指令和 PC 寄存器对应的内容错开一个周期。图 1-105说明了 PC 和指令寄存器的时序关系。

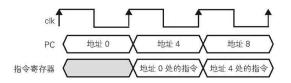


图 1-105 PC 与指令寄存器

由于 SPM 也按照时钟上升沿同步读取动作,因此从 SPM 读取指令时还要延迟一个周期。这样,指令与 PC 寄存器的对应内容会错开两个周期。图 1-106 展示了 SPM 读取操作时的时序。

使用多个时钟的数字电路设计称为多相时钟电路。由于多相时钟电路。由于多相时钟设计会导致电路动作复杂、难以验证,所以不应过多使用。AZ Processor 只在 SPM读取时使用 180 度相位的时钟。使用 180 度相位时钟的话,SPM 访问的时序会变得紧张。由于在 180 度相位时钟名张。由于在 180 度相位时钟上升沿读取的数据,要在相位0 度时钟上升沿进行锁存,实质上要求 SPM 数据读取速度为之前的两倍。

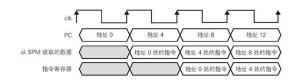


图 1-106 SPM 的读取

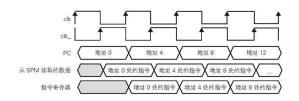


图 1-107 2 相时钟的 SPM 读取

• IF 阶段的流水线寄存器

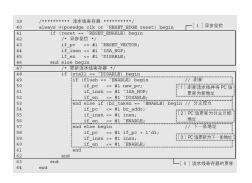
IF 阶段的流水线寄存器(if_reg)的信号线一览如表 1-44 所示,程序如代码 1-15 所示。

表 1-44 信号线一览 (if_reg.v)

分组	信号名	信号类型	数据类型	位宽	含义
时钟	clk	输入端口	wire	1	时钟
复位	reset	输入端口	wire	1	异步复位
读取数据	insn	输入端口	wire	32	读取的指

					令
	stall	输入端口	wire	1	延迟
	flush	输入端口	wire	1	刷新
流水 线制 号	new_pc	输入端口	wire	30	新程序计数器值
	br_taken	输入端口	wire	1	分支成立
	br_addr	输入端口	wire	30	分支目标地址
	if_pc	输出端口	reg	30	程序计数器
IF/ID 流水	if_insn	输出端口	reg	32	指令
流线寄	if_en	输入端口	reg	1	流水线数据有效标志位

代码 **1-15 IF** 阶段的流 水线寄存器(**if_reg.v**)



[] 异步复位

复位信号(reset)有效时寄存器将被初始化。 PC(if_pc)设置为复位向量(地址 0),指令寄存器(if_insn)设置为NOP,流水线数据有效标志位(if_en)设置为无效。

[Ⅱ]流水线寄存器的 更新

流水线寄存器在延迟信号 (stall)无效时才能更 新。

(1)处对流水线寄存器进行刷新操作。刷新信号(flush)有效时,

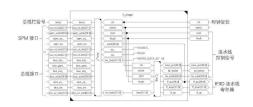
PC(if_PC)设置为新地址(new_pc),指令寄存器(if_insn)设置为NOP,流水线数据有效标志位(if_en)设置为无效。

- (2)处对分支进行处理。分支信号(br_taken)有效时,PC(if_pc)被设置为分支目的地址(br_addr)。指令寄存器(if_insn)设置为读取的指令(insn)、流水线数据有效标志位(if_en)设置为有效。
- (3) 处对 PC 的步进进行处理。在既没发生延迟也没发生分支的情况下,PC(if_pc)更新为下一条指令的地址(if_pc+1'd1)。指令寄存器(if_insn)设置为读取的指令(insn)、流水线数据有效标志位(if_en)设置为有效。

• IF 阶段的顶层模块

(as_) 设置为有效

(ENABLE_) 。



图**1-108** 端口连接图 (**if_stage.v**)

• Instruction Decode (ID) 阶段

ID 阶段对指令进行解码并生成必要的信号。数据的直通、Load 冒险的检测、分支的判定都在这一阶段进行。ID 阶段由指令解码器和流水线寄存器构成。表 1-45 列出了 ID 阶段的模块一览。

表 1-45 ID 阶段模块一览

模块名	文件名	说明
id_stage	id_stage.v	ID 阶段项层 模块
decoder	decoder.v	指令解码器
id_reg	id_reg.v	ID 阶段流水 线寄存器

• 指令解码器

指令解码器从输入的指令码中分解出各个指令字段,生成地址、数据和控制等信号。数据的直通、Load 冒险的检测、分支的判定也在这个指令解码

器中进行。表 1-46 为指令解码器的信号线一览。

表 1-46 信号线一览 (decoder.v)

分组	信号名	信号类型	数据类型
	if_pc	输出端口	reg
IF/ID 流水 线寄	if_insn	输出端口	reg
存器	if_en	输入端口	reg
	gpr_rd_data_0	输入端口	wire
GPR	gpr_rd_data_1	输入端口	wire
接口	gpr_rd_addr_0	输出端口	wire
	gpr_rd_addr_1	输出端口	wire
	id_en	输入端口	wire
1			

来自ID阶	id_dst_addr	输入端口	wire
段的 数据 直通	id_gpr_we_	输入端口	wire
	id_mem_op	输入端口	wire
	ex_en	输入端口	wire
来自 EX 阶段	ex_dst_addr	输入端口	wire
的数 据直 通	ex_gpr_we_	输入端口	wire
	ex_fwd_data	输入端口	wire
来自 MEM 阶的数直 通	mem_fwd_data	输入端口	wire
	exe_mode	输入端口	wire
控制 寄存器 日	creg_rd_data	输入端口	wire
	creg_rd_addr	输出端	wire

		П	
	alu_op	输出端口	reg
	alu_in_0	输出端口	reg
	alu_in_1	输出端口	reg
	br_addr	输出端口	reg
	br_taken	输出端口	reg
	br_flag	输出端口	reg
解码结果	mem_op	输出端口	reg
	mem_wr_data	输出端口	wire
	ctrl_op	输出端口	reg
	dst_addr	输出端口	reg
	gpr_we_	输出	reg

		端口	
	exp_code	输出端口	reg
	ld_hazard	输出端口	reg
	ор	内部信号	wire
	ra_addr	内部信号	wire
指令字段	rb_addr	内部信号	wire
	rc_addr	内部信号	wire
	imm	内部信号	wire
立即	imm_s	内部信号	wire
数	imm_u	内部信号	wire
	ra_data	内部信号	reg

从用存器	s_ra_data	内部信号	wire signec
读的据	rb_data	内部信号	reg
	s_rb_data	内部信号	wire signec
	ret_addr	内部信号	wire
地址	br_target	内部信号	wire
	jr_target	内部信号	wire

首先,指令字段的分解和 必要信号的生成部分程序 如代码 1-16 所示。

代码 1-16 内部信号生

成与输出赋值 (decoder.v)

64	/****** 指令字段 *******/	[1]指令字段的分解
65	wire ['IsaOpBus] op = if_insn['IsaOpLoc];	// 操作码
66	wire ['RegAddrBus] ra_addr = if_insn['IsaRaAddrLoc];	// Ra地址
67	wire ['RegAddrBus] rb_addr = if_insn['IsaRbAddrLoc];	// Rb地址
68	wire ['RegAddrBus] rc addr = if insn['IsaRcAddrLoc];	// Rc地址
69	wire ['IsaImmBus] imm = if_insn['IsaImmLoc];	// 立即数
70	/****** 立即数 ******/	一[]立即数字段的扩充
71	// 符号扩充	
72	wire ['WordDataBus] imm_s = {{ 'ISA_EXT_W{imm['ISA_INM	MSB] } }, imm};
73	// 0扩充	
74	wire ['WordDataBus] imm_u = {{ 'ISA_EXT_W(1'b0}}, imm}	į.
75		一 [■] 寄存器读取地址
76	assign gpr_rd_addr_0 = ra_addr; // 通用寄存器读取地址0	
77	assign gpr_rd_addr_1 = rb_addr; // 通用寄存器读取地址1	
	assign creg_rd_addr = ra_addr; // 控制寄存器读取地址	
78 79	assign creg rd addr = ra addr; // 控制寄存器读取地址 /******* 通用寄存器的读取数据 *******/	—[Ⅳ]通用寄存器的读取
78 79	assign creg rd addr = ra addr; // 控制寄存器读取地址 /******* 通用寄存器的读取数据 *******/ reg [`WordDataBus] ra_data;	// 无符号Ra
78 79 80	assign creg rd addr - ra addr; // 控制寄存器读取进址 /	// 无符号Ra ta); // 有符号Ra
78 79 80 81	assign creg rd addr = ra addr; // 控制条件据读取地址 /************************************	// 无符号Ra ta); // 有符号Ra // 无符号Rb
78 79 80 81 82	assign creg rd addr - ra addr; // 控制寄存器读取进址 /	// 无符号Ra ta); // 有符号Ra // 无符号Rb
78 79 80 81 82 83	amsign creg rd addr = ra addr; // 控制條件器恢复地址 /************************************	// 无符号Ra ta); // 有符号Ra // 无符号Rb
78 79 80 81 82 83	assign creg rd addr = ra addr; // 控稿号指录收速处 	// 无符号Ra ta); // 有符号Ra // 无符号Rb ta); // 有符号Rb
78 79 80 81 82 83 84	amsign creg rd addr = ra addr; // 控制條件器恢复地址 /************************************	// 无符号Ra ta); // 有符号Ra // 无符号Rb ta); // 有符号Rb
77 78 79 80 81 82 83 84 85 86 87	assign creg rd addr = ra addr; // 控稿号指录收速处 	// 无符号Ra // 有符号Ra // 无符号Rb ta); // 有符号Rb ta); // 有符号Rb — [V] 地址的生成 // 返回地址 R_MSB:0]; // 分支目标地址

「Ⅰ〕指令字段的分解

此处从输入的指令码中分解出各个指令字段。

[Ⅱ] 立即数字段的扩充

此处将 16 位立即数扩充到 32 位。符号扩充的立即数赋给 imm_s,0 扩充的立即数赋给 imm_u。符号扩充的立即数用该立即数字段的 MSB 填充高 16 位。0 扩充则用 0 填充高16 位。

[III] 寄存器读取地址

此处对寄存器读取地址进行赋值。通用寄存器读取地址使用指令的 Ra 字段(ra_addr)和 Rb 字段(rb_addr)。控制寄存器读取地址使用 Ra 字段(ra addr)。

[Ⅳ]通用寄存器的读取数据

此处定义通用寄存器的读取数据的信号。信号定义分为无符号(ra_data、rb_data)与有符号(s_ra_data、s_rb_data)两种。有符号信号是通过用 \$signed() 处理无符号信号得到的。

[V] 地址的生成

此处生成指令解码器中使用的地址。由于延迟间隙的存在,CALL指令的返回地址为两条指令之后的地址。因为PC(if_pc)中已经存放了下一条指令的地址,返回地址(ret_addr)为PC(if_pc)中的地址加1。

分支目标地址

(br_target)代入PC值加符号扩充后的立即数(imm_s)。因为地址为30位,32位立即数只取低位的30位参与加法运算。

跳转目标地址

(jr_target)代入 Ra 寄存器(ra_data)的值。由于 跳转目的地址

(jr_target) 为字编址,

而 Ra 寄存器(ra_data) 保存的地址为字节编址, 因此只使用 Ra 寄存器 (ra_data)高位的 30 位。

下面,与数据直通相关的程序如代码 1-17 所示。

代码 1-17 数据直通 (decoder.v)

[I] Ra 寄存器的数据 直通

因为流水线前的结果会成为最新值,直通的比较按 EX 阶段、MEM 阶段的 顺序进行。

来自 EX 阶段的数据直通的产生条件为: ID/EX 流水线寄存器有效、Ra 寄存器的读取地址

(ra_addr)与寄存器写入 地址(id_dst_addr)相 等,且寄存器的写入有效 信号(id_gpr_we_)为有 效。

来自 MEM 阶段的数据直

通的产生条件为: EX/MEM 流水线寄存器 有效、Ra 寄存器的读取 地址(ra_addr)与寄存器 写入地址(ex_dst_addr) 相等,且寄存器的写入有 效信号(ex_gpr_we_)为 有效。无法进行直通时, 直接使用寄存器堆读取 值。

[II] **Rb** 寄存器的数据 直通

来自 EX 阶段的数据直通的产生条件为: ID/EX 流水线寄存器有效、Rb 寄存器的读取地址

(rb_addr)与寄存器写入地址(id_dst_addr)相等,且寄存器的写入有效信号(id_gpr_we_)为有效。来自 MEM 阶段的据直通的产生条件为: EX/ MEM 流水线寄存器的产生条件为: EX/ MEM 流水线寄存器的声类器有效、Rb 寄存器的诗界器地址(rb_addr)与寄存器的写入地址(ex_dst_addr)相等,且寄存器的写入为有效信号(ex_gpr_we_)为有效信号、无法进行数据重读取值。

Load 冒险检测程序如代码 1-18 所示。

代码 1-18 Load 冒险检

测 (decoder.v)

「I] Load 冒险检测

Load 冒险产生的条件为: ID/EX 流水线寄存器中存放的之前的指令为Load 指令,通用寄存器的写入地址与当前指令的读取地址相等。ID/EX 流水线寄存器有效、内存操作(id_mem_op)为 Load 指令

(MEM_OP_LDW),且 之前指令的写入地址

(id_dst_addr)与 Ra 寄 存器的地址(ra_addr)或 Rb 寄存器的地址

(rb_addr) 相等时使能 Load 冒险信号 (ld hazard)。

只将与默认值不同的信号 赋予新值。代码 1-19 列 出的是信号初始化部分程 序。

代码 **1-19** 内部信号初始化(**decoder.v**)



[I] 默认信号的默认 值

此处依据表 1-47 所示的 默认值进行初始化。

下面,代码 1-20 展示了 逻辑运算指令解码部分程 序。

代码 1-20 逻辑运算指令解码(decoder.v)



表1-47 解码结果

-	-	ALU
ANDR	ISA_OP_ANDR	ALU
ANDI	ISA_OP_ANDI	ALU
ORR	ISA_OP_ORR	ALU
ORI	ISA_OP_ORI	ALU
XORR	ISA_OP_XORR	ALU
XORI	ISA_OP_XORI	ALU
ADDSR	ISA_OP_ADDSR	ALU
ADDSI	ISA_OP_ADDSI	ALU
ADDUR	ISA_OP_ADDUR	ALU
ADDUI	ISA_OP_ADDUI	ALU
SUBSR	ISA_OP_SUBSR	ALU
SUBUR	ISA_OP_SUBUR	ALU
SHRLR	ISA_OP_SHRLR	ALU
SHRLI	ISA_OP_SHRLI	ALU
SHLLR	ISA_OP_SHLLR	ALU
SHLLI	ISA_OP_SHLLI	ALU
BE	ISA_OP_BE	ALU
BNE	ISA_OP_BNE	ALU
BSGT	ISA_OP_BSGT	ALU
BUGT	ISA_OP_BUGT	ALU
JMP	ISA_OP_JMP	ALU
CALL	ISA_OP_CALL	ALU
LDW	ISA_OP_LDW	ALU
STW	ISA_OP_STW	ALU
TRAP	ISA_OP_TRAP	ALU
RDCR	ISA_OP_RDCR	ALU
WRCR	ISA_OP_WRCR	ALU
EXRT	ISA_OP_EXRT	ALU

[I] **ANDR** 指令解码

此处将 ALU 操作
(alu_op) 设置为
AND(ALU_OP_AND),
通用寄存器写入地址
(dst_addr)中记入 Rc 寄存器(rc_addr),通用寄存器写入有效信号
(gpr_we_)设置为有效。

[II] ANDI 指令解码

此处将 ALU 操作 (alu_op)设置为 AND(ALU_OP_AND), ALU 的 1 号输入 (alu_in_1)代入 0 扩充 后的立即数(imm_u), 通用寄存器写入有效信号 (gpr_we_)设置为有 效。

[III] ORR 指令解码

此处将 ALU 操作
(alu_op) 设置为
OR (ALU_OP_OR),通
用寄存器写入地址
(dst_addr)中记入 Rc 寄存器 (rc_addr),通用寄存器写入有效信号
(gpr_we_)设置为有效。

「IV] ORI 指令解码

此处将 ALU 操作(alu_op)设置为

OR(ALU_OP_OR), ALU 的 1 号输入 (alu_in_1)代入 0 扩充 后的立即数(imm_u), 通用寄存器写入有效信号 (gpr_we_)设置为有 效。

「V】**XORR** 指令解码

此处将 ALU 操作
(alu_op)设置为
XOR(ALU_OP_XOR),
通用寄存器写 入地址
(dst_addr)中记入 Rc 寄
存器(rc_addr),通用寄
存器写入有效信号
(gpr_we_)设置为有
效。

[VI] **XORI** 指令解码

此处将 ALU 操作
(alu_op)设置为
XOR(ALU_OP_XOR),
ALU 的 1 号输入
(alu_in_1)代入 0 扩充
后的立即数(imm_u),
通用寄存器写入有效信号
(gpr_we_)设置为有
效。

接下来,我们对算术运算 指令的解码程序进行说 明,如代码 1-21 所示。

代码 **1-21** 算术运算指 令解码(**decoder.v**)

172	/* 算术运算指令 */	
173	`ISA_OP_ADDSR : begin // 寄存器间的	有符号加法
174	alu_op = `ALU_OP_ADDS;	
175	dst_addr = rc_addr;	[I]ADDSR指令解码
176	gpr_we_ = `ENABLE_;	
177	end	
178	'ISA OP ADDSI : begin // 寄存器与立	即数间的有符号加法
179	alu op = 'ALU OP ADDS;	
180	alu in 1 = imm s;	[II]ADDSI指令解码
181	gpr_we_ = `ENABLE_;	1
182	end	
183	`ISA_OP_ADDUR : begin // 寄存器间的	无符号加法
184	alu_op = `ALU_OP_ADDU;	
185	dst_addr = rc_addr;	[III]ADDUR 指令解码
186	gpr_we_ = `ENABLE_;	
187	end	
188	`ISA_OP_ADDUI : begin // 寄存器与立	即数间的无符号加法
189	'ISA_OP_ADDUI : begin // 希伊器与证 alu_op = 'ALU_OP_ADDU; alu_in_1 = imm_s;	
190		[IV] ADDUI 指令解码
191	gpr_we_ = `ENABLE_;	
192	end	
193	`ISA_OP_SUBSR : begin // 寄存器间的	有符号减法
194	alu_op = `ALU_OP_SUBS;	
195	dst_addr = rc_addr;	[V]SUBSR 指令解码
196	gpr_we_ = `ENABLE_;	
197	end	
198	`ISA_OP_SUBUR : begin // 寄存器间的	无符号减法
199	"ISA_OP_SUBUR : begin // 崇存器间的 alu_op = "ALU_OP_SUBU; dst_addr = rc_addr;	5 to 5 at 100 at 10 at 10 at 10
200	dst_addr = rc_addr;	[VI] SUBUR 指令解码
201	gpr_we_ = `ENABLE_;	
202	end	

[I] ADDSR 指令解码

此处将 ALU 操作 (alu_op)设置为有符号 加法 (ALU_OP_ADDS),通 用寄存器写入地址 (dst_addr)中记入 Rc 寄 存器(rc_addr),通用寄 存器写入有效信号 (gpr_we_)设置为有 效。

[II] ADDSI 指令解码

此处将 ALU 操作 (alu_op) 设置为有符号 加法 (ALU_OP_ADDS), ALU 的 1 号输入 (alu_in_1) 代入符号扩充后的立即数 (imm_s),通用寄存器 写入有效信号 (gpr_we_) 设置为有效。

[III] ADDUR 指令解码

此处将 ALU 操作
(alu_op) 设置为无符号
加法
(ALU_OP_ADDU),
通用寄存器写入地址
(dst_addr) 中记入 Rc 寄存器(rc_addr), 通用寄存器写入有效信号
(gpr_we_) 设置为有效。

[IV] ADDUI 指令解码

此处将 ALU 操作 (alu_op) 设置为无符号 加法 (ALU_OP_ADDU), ALU 的 1 号输入 (alu_in_1) 代入符号扩 充后的立即数 (imm_s),通用寄存器 写入有效信号 (gpr_we_) 设置为有 效。

[V] **SUBSR** 指令解码

此处将 ALU 操作 (alu_op)设置为有符号 减法 (ALU_OP_SUBS),通 用寄存器写入地址 (dst_addr)中记入 Rc 寄 存器(rc_addr),通用寄 存器写入有效信号 (gpr_we_)设置为有 效。

[VI] SUBUR 指令解码

此处将 ALU 操作 (alu_op) 设置为无符号 减法 (ALU OP SUBU),通

(ALU_OP_SUBU), 題 用寄存器写入地址 (dst_addr)中记入 Rc 寄 存器 (rc_addr), 通用寄 存器写入有效信号 (gpr_we_)设置为有 效。

接下来,我们对移位指令的解码程序进行说明,如 代码 1-22 所示。

代码 **1-22** 移位指令解码(**decoder.v**)

203	/* 移位指令 */		
204	`ISA OP SHRLR : begin // 寄存器间的逻辑右移		
205	alu_op = `ALU_OP_SHRL;		
206	dst_addr = rc_addr;	I I ISHRL	R指令解码
207	gpr_we_ = `ENABLE_;	1810 500000	
208	end		
209	`ISA_OP_SHRLI : begin // 寄存器与立即数间的	逻辑右移	
210	alu_op = `ALU_OP_SHRL;		
211	alu in 1 = imm u;	I II I SHRL	指令解码
212	gpr_we_ = `ENABLE_;		
213	end		
214	'ISA_OP_SHLLR : begin // 寄存器间的逻辑左移		
215	alu_op = `ALU_OP_SHLL;		
216	dst_addr = rc_addr;	[II]SHLLI	R指令解码
217	gpr_we_ = `ENABLE_;		
218	end		
219	"ISA_OP_SHLLI : begin // 寄存器与立即数间的	逻辑左移	
220	alu_op = `ALU_OP_SHLL;		
221	alu_in_1 = imm_u;	[IV]SHLL	指令解码
222	gpr_we_ = `ENABLE_;		
223	end		

「I] SHRLR 指令解码

此处将ALU操作
(alu_op)设置为逻辑右移(ALU_OP_SHRL)、通用寄存器写入地址
(dst_addr)中记入Rc寄存器(rc_addr)、通用寄存器写入有效信号
(gpr_we_)设置为有效。

[II] SHRLI 指令解码

此处将 ALU 操作 (alu_op)设置为逻辑右 移(ALU_OP_SHRL), ALU 的 1 号输入 (alu_in_1)代入 0 扩充 后的立即数(imm_u), 通用寄存器写入有效信号 (gpr_we_)设置为有 效。

[III] SHLLR 指令解码

此处将 ALU 操作
(alu_op)设置为逻辑左
移(ALU_OP_SHLL),
通用寄存器写入地址
(dst_addr)中记入 Rc 寄存器(rc_addr),通用寄存器写入有效信号
(gpr_we_)设置为有效。

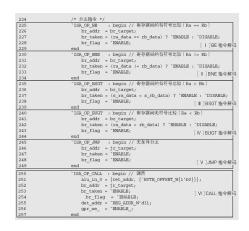
「IV] SHLLI 指令解码

此处将 ALU 操作 (alu_op) 设置为逻辑左 移(ALU_OP_SHLL), ALU 的 1 号输入 (alu_in_1)代入 0 扩充 后的立即数(imm_u), 通用寄存器写入有效信号 (gpr_we_)设置为有 效。

接下来,我们对分支指令的解码程序进行说明,如

代码 1-23 所示。

代码 **1-23** 分支指令解码(**decoder.v**)



[I] **BE** 指令解码

此处将分支目标地址(br_target)输出给分支地址(br_addr),并设置分支符号位(br_flag)为有效。Ra 寄存器(ra_data)与Rb寄存器(rb_data)相等时,分支成立信号(br_taken)有效。

[Ⅱ] **BNE** 指令解码

此处将分支目标地址(br_target)输出给分支地址(br_addr),并设置分支符号位(br_flag)为有效。Ra寄存器(ra_data)与Rb寄存器(rb_data)不等时,分支成立信号(br_taken)有效。

[III] BSGT 指令解码

此处将分支目标地址(br_target)输出给分支地址(br_addr),并设置分支符号位(br_flag)为有效。Rb寄存器(s_rb_data)比Ra寄存器(s_ra_data)大时,分支成立信号(br_taken)有效。因为BSGT指令为有符号比较,对寄存器进行比较时,使用有符号信号。

[IV] BUGT 指令解码

此处将分支目标地址(br_target)输出给分支地址(br_addr),并设置分支符号位(br_flag)为有效。Rb寄存器(rb_data)比Ra寄存器(ra_data)大时,分支成立信号(br_taken)有效。

[V] JMP 指令解码

此处将分支目标地址(jr_target)输出给分支地址(br_addr),并设置分支符号位(br_flag)为有效。由于 JMP 指令为无条件跳转,分支成立信号(br_taken)总 是有效。

[VI] CALL 指令解码

此处将分支目标地址 (jr_target) 输出给分支 地址(br_addr),并设置 分支符号位(br flag)为 有效。由于 CALL 指令为 无条件跳转,分支成立信 号(br taken)总是有 效。因为要将 CALL 指令 的返回地址(ret addr) 写入31号通用寄存器, 返回地址(ret addr)要 代入ALU的0号输入 (alu in 0)。然后将通 用寄存器写入地址 (dst addr) 指定为31号 通用寄存器的地址, 并使 能通用寄存器写入有效信 号(gpr_we_)。由于返 回地址 (ret addr) 为 30 位的字编址格式,最低两 位用 0 扩充, 然后代入 ALU 的 0 号输入 (alu in 0) .

接下来,我们对内存访问 指令的解码程序进行说 明,如代码 1-24 所示。

代码 **1-24** 内存访问指 令解码(**decoder.v**)

258	/* 内存访问指令 */	
259	`ISA_OP_LDW : begin // 字读取	
260	alu_op = 'ALU_OP_ADDU;	
261	alu_in_1 = imm_s;	[I]LDW 指令解
262	mem_op = `MEM_OP_LDW;	
263	gpr_we_ = `ENABLE_;	
264	end	
265	`ISA_OP_STW : begin // 字写入	
266	alu_op = 'ALU_OP_ADDU;	
267	alu_in_1 = imm_s;	[II]STW 指令解
268	mem_op = `MEM_OP_STW;	
269	end	

[I] **LDW** 指令解码

为了进行地址计算,需要将 ALU 操作(alu_op)设置为无符号加法(ALU_OP_ADDU),并将符号扩充后的立即数(imm_s)代入 ALU 的 1号输入(alu_in_1)。内存操作(mem_op)设置为字读取(MEM_OP_LDW),并使能通用寄存器写入有效信号(gpr_we_)。

[II] STW 指令解码

为了进行地址计算,需要将 ALU 操作(alu_op)设置为无符号加法(ALU_OP_ADDU),并将符号扩充后的立即数(imm_s)代入 ALU 的 1号输入(alu_in_1)。内存操作(mem_op)设置为字写入(MEM OP STW)。

接下来,我们对特殊指令的解码程序进行说明,如 代码 1-25 所示。

代码 **1-25** 特殊指令解码(**decoder.v**)

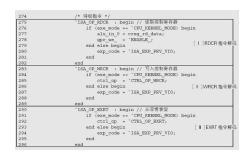


「I] TRAP 指令解码

TRAP 指令是引发陷阱异常的指令,因此将陷阱异常的异常代码(ISA_EXP_TRAP)代入异常代码信号(exp code)中。

接下来,我们对特权指令的解码程序进行说明,如 代码 1-26 所示。

代码 **1-26** 特权指令解码(**decoder.v**)



「I] RDCR 指令解码

此处将从控制寄存器读取的值(creg_rd_data)代入 ALU 的 1 号输入,并使能通用寄存器写入有效信号(gpr_we_)。特权指令在内核模式之外模式执行时会引发特权异常。异常代码信号

(exp_code)代入特权违 反异常的异常代码 (ISA_EXP_PRV_VIO)。

[II] WRCR 指令解码

此处将控制操作

(ctrl_op) 设置为写入 (CTRL_OP_WRCR)。

[III] EXRT 指令解码

此处将控制操作 (ctrl_op)设置为异常恢 复操作 (CTRL OP EXRT)。

最后,当读入未定义指令时的处理程序如代码 1-27 所示。

代码 **1-27** 未定义指令的处理(**decoder.v**)



[I] 未定义指令的处理

当读入未定义的指令时, 在此处引发未定义指令异 常。异常代码信号 (exp_code)代入未定义 指令的异常代码 (ISA_EXP_UNDEF_INSI

• ID 阶段流水线寄存器

ID 阶段流水线寄存器(id_reg)的信号线一览如表 1-48 所示,程序如代码 1-28 所示。

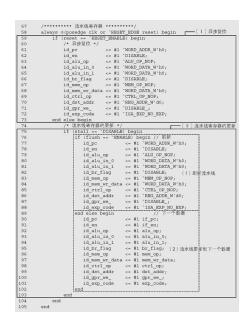
表 1-48 信号线一览 (id_reg.v)

分组	信号名	信号类型	
时钟	clk	输入端口	wi
复位	reset	输入端口	wi
	alu_op	输入端口	wi
	alu_in_0	输入端口	wi
	alu_in_1	输入端口	wi
	br_flag	输入端口	wi
	mem_op	输入端口	wi
解码	mem_wr_data	输入端口	wi
	ctrl_op	输入端口	wi
	dst_addr	输入端口	wi

	gpr_we_	输入端口	wi
	exp_code	输入端口	wi
流水 线控信 号	stall	输入端口	wi
	flush	输入端口	wi
IF/ID 流水	if_pc	输入端口	wi
线寄存器	if_en	输入端口	wi
	id_pc	输出端口	reş
	id_en	输出端口	reş
	id_alu_op	输出端口	reş
	id_alu_in_0	输出端口	reş
		输出	_

	id_alu_in_1	端口	reş
ID/EX 流水 线寄 存器	id_br_flag	输出端口	reş
	id_mem_op	输出端口	reş
	id_mem_wr_data	输出端口	reş
	id_ctrl_op	输出端口	reş
	id_dst_addr	输出端口	reş
	id_gpr_we_	输出端口	reş
	id_exp_code	输出端口	reş

代码 **1-28 ID** 阶段流水 线寄存器(**id_reg.v**)



[] 异步复位

复位信号(reset)有效时寄存器会被初始化。因为复位时流水线内的数据无效,初始化时,此处将全部控制信号设为无效,数据信号设为0。

[II]流水线寄存器的 更新

流水线寄存器在延迟信号 (stall)无效时才可更 新。(1)处执行流水线 寄存器的刷新操作。刷新 信号(flush)有效时, 有流水线寄存器都会被 始化。(2)处执行流水 线寄存器的更新操作。将 指令解码的结果存入流水 线寄存器。

• **ID** 阶段顶层模块

ID 阶段项层模块用来连接指令解码器与ID 阶段流水线寄存器。图 1-109展示了 ID 阶段项层模块的连接图。



图1-109 端口连接图 (id_stage.v)

• Execution (EX) 阶段

EX 阶段主要进行运算和中断检测操作。EX 阶段由算术逻辑运算单元和流水线寄存器构成。表 1-49为 EX 阶段模块一览。

表 **1-49 EX** 阶段模块一览

模块名	文件名	说明
ex_stage	ex_stage.v	EX 阶 段顶层 模块
alu	alu.v	算术逻 辑运算 单元
ex_reg	ex_reg.v	EX 阶 段流水 线寄存 器

• ALU

ALU 根据输入指定的操作对数据进行处理,并输出处理结果。ALU 的输入为一个操作码和两个数据,输出为运算结果和溢出信号。ALU 的框图如图 1-110 所示,信号线一览如表 1-50 所示,源程序如代码 1-29 所示。

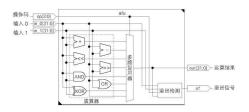


图 1-110 ALU 模块图

表 **1-50** 信号线一览 (alu.v)

分组	信号名	信号类型	数据类型	位宽	含义
	in_0	输入端口	wire	32	输 入 0
输入	in_1	输入端口	wire	32	输 入 1
	op	输入端口	wire	4	操作
运算结	out	输出端口	reg	32	输出
当					

果	of	输出端口	reg	1	溢出
	s_in_0	内部信号	wire signed	32	有符号输入0
内部信号	s_in_1	内部信号	wire signed	32	有符号输入1
	s_out	内部信号	wire signed	32	有符号输出

代码 1-29 ALU(alu.v)

28	/********** 有符号输入输出信号 ********/ [1] 有符号信号的生成
29	/******* 有符号输入输出信号 ********/ wire signed [`WordDataBus] s in 0 = \$signed(in 0); // 有符号输入0
30	wire signed [wordDataBus] s_in_0 = \$signed(in_0); // 有符号输入0 wire signed [WordDataBus] s_in_1 = \$signed(in_1); // 有符号输入1
31	wire signed ['WordDataBus] s out = \$signed(out); // 有符号输出
32	Haze dagmon (Horadocomo) d_dec - parginar(dec), // // // dag
33	/****** 算术逻辑运算 *******/ [] 算术逻辑运算
34	always @(*) begin
35	case (op)
36	"ALU_OP_AND : begin // 逻辑与(AND)
37	out = in_0 & in_1; (1)逻辑与(AND)
38	end
39	ALU_OP_OR : begin // 逻辑或(OR) out = in 0 in 1; (2) 逻辑或(OR)
41	out = in_0 in_1; (2)逻辑或(OR) end
42	`ALU_OP_XOR : begin // 逻辑异或 (XOR)
43	out = in_0 'in_1; (3)逻辑异或(xor)
44	end - 10_1 10_1/
45	"ALU_OP_ADDS : begin // 有符号加法
46	out = in 0 + in 1; (4) 有符号加法
47	end
48	`ALU_OP_ADDU : begin // 无符号加法
49	out = in_0 + in_1; (5) 无符号加法
50	end
51	`ALU_OP_SUBS : begin // 有符号减法
52	out = in_0 - in_1; (6) 有符号減法
53	end
54 55	`ALU_OP_SUBU : begin // 无符号减法
	out = in_0 - in_1; (7) 无符号减法
56 57	end `ALU OP SHRL : begin // 逻辑右移
58	out = in_0 >> in_1[`ShAmountLoc]; (8) 逻辑右移
59	end (0) Activity
60	`ALU_OP_SHLL : begin // 逻辑左移
61	out = in 0 << in 1[`ShAmountLoc]; (9) 逻辑左移
62	end
63	default : begin // 默认值 (No Operation)
64	out = in_0; (10) 默认值(No Operation)
65	, end
66	endcase
67	end
69	/****** 溢出检测 ******/ [I] 溢出检测
70	always @(*) begin
71	case (op)
72	`ALU_OP_ADDS : begin // 加法溢出检测
73	if (((s_in_0 > 0) && (s_in_1 > 0) && (s_out < 0))
74	((s_in_0 < 0) && (s_in_1 < 0) && (s_out > 0))) begin
75	of = `ENABLE;
	end else begin
76	
77	of = `DISABLE;
77 78	end (Carl Management Approximation Approxima
77	end ((11) 加波溢出档測
77 78 79 80	end end [[11]]加法溢出绘測 ["ALU_OP_SUBS : begin // 高法溢出检测
77 78 79 80 81	end ([11],加速過出控制 TALU OF SUBS : begin // 減速過控制 if (((e in 0 < 0) && (e in 1 > 0) && (e out > 0))
77 78 79 80 81 82	end end (111)加速油時機測 ALM_OP_SUBS: begin // 減速溢出物類
77 78 79 80 81 82 83	end end *AU 00_SUBS : begin // \$\frac{2}{3}
77 78 79 80 81 82 83 84	end end "ALL/OP_SUBS: begin // 淡淡溢出热潮 if (((e_in_0 < 0) && (e_in_1 > 0) && (e_out > 0))
77 78 79 80 81 82 83 84 85	end end **MU 00_smms : begin // ##################################
77 78 79 80 81 82 83 84 85 86	end end *AUJ OP SUBS: begin // 淡淡溢色热潮 if ((e,in,0 < 0) && (e,in,1 > 0) && (e,out > 0)) ((e,in,0 < 0) && (e,in,1 < 0) && (e,out < 0))) begin of "EMBALE; end else begin of "DISABLE; end
77 78 79 80 81 82 83 84 85 86 87	end end *AU_or_coms: begin // 東京報告批判 if ((e_in_o < 0 & (e_in_i > 0) & (e_out > 0)) or * "BMAIS & (e_in_i < 0) & (e_out < 0)) begin of * "BMAIS & (e_in_i < 0) & (e_out < 0)) begin of * "DISABLE; end end
77 78 79 80 81 82 83 84 85 86 87	end end **ALI OF SUBS : begin // 高麗祖田歌劇 if ((e_in_0 < 0) && (e_in_1 > 0) && (e_out < 0))
77 78 79 80 81 82 83 84 85 86 87 88	end end **AULOP_SUBE : begin // 東海田田原 **If ((e_in_0 < 0) && (e_in_0 < 0) && (e_out < 0))
77 78 79 80 81 82 83 84 85 86 87	end end **ALI OF SUBS : begin // 高麗祖田歌劇 if ((e_in_0 < 0) && (e_in_1 > 0) && (e_out < 0))

[I] 有符号信号的生成

此处将输入信号(in_0, in_1)与输出信号(out) 生成为有符号信号。有符 号信号将被用在有符号加 法和减法的溢出检测中。

[II] 算术逻辑运算

此处进行以下 9 种运算操作: (1)逻辑与 (AND)、(2)逻辑或 (OR)、(3)逻辑异或 (XOR)、(4)有符号 加法、(5)无符号加法、(6)有符号减法、(6)无符号减法、(8)逻辑右移、(9)逻辑左

32 位的移位运算,最大 位移量为 32 位。因此 (8) 和(9) 的移位运算 中,右边第二项输入使用 5 位

(in_1[`ShAmountLoc])。 5 位可以表达的最大值为 2 的 5 次方,即 32。不进 行任何运算(No Operation)时,在(10) 处直接输出输入 0 的值。

ALU 的 NOP 在 CALL、WRCR、RDCR 等指令执行时,为了将 ID 阶段读取的寄存器的值按原样写回时使用。

[III] 溢出检测

在进行有符号加法和减法运算时,需要检测溢出。 因此,(11)、(12)处 分别对加法和减法的溢出 进行检测。

加法溢出发生的条件为: 正数加正数结果为负数, 或负数加负数结果为正 数。在处理有符号加法 后,在(11)处检测该条 件,如果条件满足则使能 溢出信号(of)。

减法溢出发生的条件为: 负数减去正数结果为正

数,或正数减去负数结果为负数。在处理有符号减法后,在(12)处检测该条件,如果条件满足则使能溢出信号(of)。在处理有符号加法、减法以外的运算时,在(13)处设置溢出信号(of)为无效。

• **EX** 阶段流水线寄存器

EX 阶段流水线寄存器的信号线一览如表 1-51 所示,源程序如代码 1-30 所示。

表 **1-51** 信号线一览 (ex_reg.v)

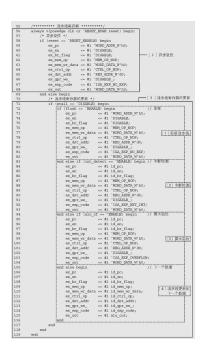
分组	信号名
时钟复位	clk
	reset
ALU 的 输出	alu_out
	alu_of

	stall
流水线控制 信号	flush
	int_detect
	id_pc
	id_en
	id_br_flag
	id_mem_op
ID/EX 流 水线 寄	id_mem_wr_dat
存器	id_ctrl_op

	id_dst_addr
	id_gpr_we_
	id_exp_code
	ex_pc
	ex_en
	ex_br_flag
	ex_mem_op

	ex_mem_wr_da
EX/MEM 流水线寄 存器	ex_ctrl_op
	ex_dst_addr
	ex_gpr_we_
	ex_exp_code
	ex_out

代码 **1-30 EX** 阶段 流水线寄存器 (**ex_reg.v**)



[I] 异步复位

复位信号(reset)有效时寄存器会被初始化。因为复位时流水线内的数据无效,初始化时,此处将全部控制信号设为无效,数据信号设为 0。

[Ⅱ]流水线寄存器 的更新

流水线寄存器在延迟 信号(stall)无效时 才可更新。

(1) 处对流水线寄存器进行刷新操作。 当刷新信号(flush) 有效时,所有流水线 寄存器将被初始化。

- (2) 处对中断进行 检测。如果中断检测 信号(int detect)有 效,则中止正在执行 的指令,并将异常代 码 (ex_exp_code) 设置为外部中断异常 (ISA EXP EXT IN: 中止指令操作时,将 内存操作信号 (ex mem op)、控 制寄存器操作信号 (ex ctrl op) 和通 用寄存器写入有效信 号 (ex_gpr_we_) 设 置为无效。同时,将 内存写入数据 (ex mem wr data) 通用寄存器写入地址 (mem dst addr) 和 处理结果 (ex out) 设置为0。
- (3) 处对溢出异常进行检测。如果溢出信号(alu_of)有效,则中止正在执行的指令操作,并将异常代码
- (ex_exp_code) 设 置为溢出异常
 - (ISA_EXP_OVERFL
- (4) 处对流水线寄存器进行更新。运算处理的结果在此处被存储到流水线寄存

器。

• EX 阶段顶层模块

EX 阶段项层模块用来连接 ALU 与 EX 阶段流水线寄存器。图 1-111 展示了 EX 阶段项层模块的连接图。

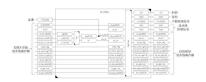


图1-111 端口连接 图 (ex_stage.v)

• Memory (MEM) 阶段

MEM 阶段主要负责内存的访问。在执行 LDW 和STW 等指令时,内存访问操作是在 MEM 阶段进行的。MEM 阶段由内存访问控制模块、流水线寄存器、以及总线接口构成。表 1-52 为 MEM 阶段的模块一览。

表 **1-52 MEM** 阶段模块 一览

模块名	文件名	说明
mem_stage	mem_stage.v	MEM 阶段 顶层 模块
		内存

mem_ctrl	mem_ctrl.v	访问 控制 模块
mem_reg	mem_reg.v	MEM 阶段 流水 线寄 存器
bus_if	bus_if.v	总线 接口

• 内存访问控制模块

内存访问控制模块基于从 EX 阶段流水线寄存器输入的内存操作(ex_mem_op),实施内存访问操作。内存访问控制模块的信号线一览如表 1-53 所示,源程序如代码1-31 所示。

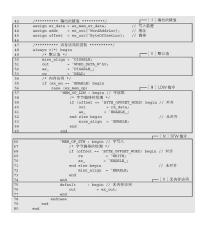
表 **1-53** 信号线一览 (mem_ctrl.v)

分组	信号名
	ex_en
EX/MEM 流水线寄	ex_mem_op

存器	
	ex_mem_wr_da
	ex_out
	rd_data
	addr
内存访问 接口	as_
	rw
	wr_data
	out
内存访问 结果	miss_align
	offset

代码 **1-31** 内存访问 控制模块

(mem_ctrl.v)



[I] 输出的赋值

此处进行一系列输出的赋值: EX 阶段的写入数据

(ex_mem_wr_data) 代入写入数据

(wr_data), EX 阶段输出(ex_out)的高 30 位代入地址(addr), EX 阶段输出(ex_out)的低2 位代入字节偏移

[II] 默认值

(offset) 。

地址选通信号 (as_)默认设置为 无效,读/写信号默

认设置为读取

(READ),输出信号(out)默认设置为0。

[III] LDW 指令

LDW 指令执行时, 需要对地址是否按字 对齐进行检测。字节 偏移(offset)为 0 (BYTE OFFSET V 时,地址是对齐的, 因此直接使能地址选 通信号。LDW 为读 取访问指令,要将读 取数据(rd_data)赋 值到输出(out)。 字节偏移 (offset) 不为 0 (BYTE OFFSET V 时,地址未对齐,使 能未对齐信号 (miss align) .

「IV] STW 指令

STW 指令执行时,需要对地址是否按字对齐进行检测。字节偏移(offset)为 0(BYTE_OFFSET_V时,地重接使能地址是对齐的,选通信号。字节偏移(offset)不为 0(BYTE_OFFSET_V时,地址未对齐,使能未对齐信号

(miss_align) 。

[V] 无内存访问

在没有内存访问操作 发生时,直接将 MEM 阶段的输出 (ex_out)赋值给输 出(out)。

• MEM 阶段流水线寄存器

MEM 阶段流水线寄存器的信号线一览如表 1-54 所示,源程序如代码 1-32 所示。

表 1-54 信号线一览 (mem_reg.v)

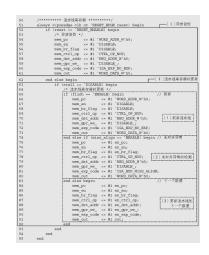
1	
分组	信号名
时钟复位	clk
	reset
内存访问结果	out
	miss_align

流水线控	stall
制信号	flush
	ex_pc
EX/MEM 流水线寄 存器	ex_en
	ex_br_flag
	ex_ctrl_op
	ex_dst_addr

	ex_gpr_we_
	ex_exp_code
	mem_pc
	mem_en
	mem_br_flag
MEM/W 流水线	
存器	mem_dst_addr

mem_gpr_we_
mem_exp_code
mem_out

代码 **1-32 MEM** 阶 段流水线寄存器 (**mem_reg.v**)



[I] 异步复位

复位信号(reset)有效时寄存器会被初始化。因为复位时流水线内的数据无效,初始化时,此处将全部控制信号设为无效,

数据信号设为0。

[II]流水线寄存器 的更新

流水线寄存器在延迟 信号(stall)无效时 才可更新。

- (1) 处对流水线寄存器进行刷新操作。 当刷新信号(flush) 有效时,所有流水线 寄存器将被初始化。
- (2) 处对未对齐异常进行检测。未对齐信号(miss_align)有效时,中止正在进行的操作,将异常代码

(mem_exp_code) 设置为未对齐异常 (ISA_EXP_MISS_A] 中止指令操作时,控 制寄存器操作信号 (ex_ctrl_op)、通 用寄存器写入有效信 号(ex_gpr_we_)设置为无效。同时,将 通用寄存器写入地址 (mem_dst_addr)、 处理结果 (mem_out)设置为 0。

(3)处对流水线寄存器进行更新。内存

操作的结果在此处被 存储到流水线寄存 器。

• MEM 阶段顶层模块

MEM 阶段顶层模块用来连内存访问控制模块、MEM 阶段流水线寄存器、与总线接口。图 1-112 展示了 MEM 阶段顶层模块的端口连接图。

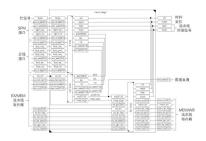


图 1-112 端口连接 图(mem_stage.v)

• CPU 控制模块

CPU 控制模块进行对保存 CPU 状态的控制有关的控制等存器进行管理,并对流水线进行控制,并对流水线进行控制模的控制。CPU 控制模的模块中设有设置和保存。是U 状态的控制存存的控制。表 1-55 为 CPU 宏。表 1-55 为 CPU 控制寄存器的一览。

表 1-55 CPU 控制

寄存器

		特科类型		*		and dele	ID B	40
0	状态	RW	- Inteleded edede	Beserve		11111111		10
ij	前一个状态	RW		Reserve	d		T,	6 0
2	经净计数器	R		PC			-	9 0
3	异常程序计数器	RW.		EPC			-	0
4	非常向量	RW		EXP_VECT	OR RO		-	3 0
5	异苯聚四杏苷基	RW		Reserved			co	30
6	中原蔣藪	RW		Reserved		MAS	k.	
7	中原領家	B		Reserved		IRQ		
0 - 20	98	-		Rese	rved			
29	ROM SE	R		ROM_	SIZE			
30	SPM 容量	R		SPM.	SZE			_
31	CPUES	R	YEAR	MONTH	VER	REV		

控制寄存器 0: 状态

> **[0]**: 执行模式 寄存器

(EM:Execution Mode)

用于设定 CPU 的执行模式。该位为 0 时表示 CPU 处于内核模式,为 1 时表示 CPU 处于用户模式。

[1]: 中断有效 (IE:Interrupt Enable)

设置该位时中断有效。

• 控制寄存器 1: 前一个状态

[0]: 执行模式 寄存器

(EM:Execution

Mode)

用于保存异常发生前的 CPU 执行模式。

[1]: 中断有效 (IE:Interrupt Enable)

用于保存异常发 生前的中断有效 位。

• 控制寄存器 2: 程序计数器

[31:2]: 程序计数器

(PC:Program Counter)

用于读取当前程 序计数器。

• 控制寄存器 3: 异常程序计数器

[31:2]: 异常程 序计数器 (EPC:Exception Program Counter)

用于保存异常发 生时的程序计数 器。

• 控制寄存器 4: 异常向量

[31:2]: 异常向 量

(EXP_VECTO) Vector)

用于设定异常处 理程序地址。异 常发生时跳转到 异常向量所存储 的地址。

• 控制寄存器 5: 异常原因寄存器

[**2:0**]: 异常代码

(CODE:Except Code)

用于存储所发生 异常的异常代 码。

[3]: 延迟间隙 标志位 (D:Delay Slot Flag)

发生延迟间隙异 常时,该标志位 有效。

• 控制寄存器 6: 中断屏蔽

[**7:0**]: 中断屏 蔽

(MASK:Interru

Mask)

用于设定中断屏 蔽(也称为中断 掩字)。通过设 置该寄存器,可 以屏蔽指定中 断。

• 控制寄存器 7: 中断请求

[7:0]: 中断请 求 (IRQ:Interrupt Request)

用于读取中断请求。

控制寄存器 29: ROM 容量

[31:0]: ROM 容量 (ROM_SIZE:R Size)

用于读取所用 ROM 的容量。

控制寄存器 30: SPM 容量

[31:0]: SPM 容 量 (SPM_SIZE:SF Size)

用于读取所用 SPM 的容量。

控制寄存器 31: CPU 信息

[31:24]:制作年份

(YEAR:Year)

用于读取制作年份。制作年份为 1970加该寄存器中的值。

[**23:16**]:制作月份

(MONTH:Mon

用于读取制作月份。

[15:8]: 版本号 (VER:Version)

用于读取 CPU 的版本号。

[7:0]:修订号 (REV:Revision

用于读取 CPU 的修订号。

控制寄存器 0~7 用来控制 CPU 操作或读取 CPU 状态。控制寄存器 29~31 用来读

取内存容量、CPU 版本等 CPU 相关信息。

表 1-56 为 CPU 控制 模块的信号线一览。

表 **1-56** 信号线一览 (ctrl.v)

分组	信号名
时钟	clk
复位	reset
控制寄存器接口	creg_rd_addr
	creg_rd_data
	exe_mode
中断	irq
	int_detect

ID/EX 流 水线寄存 器	id_pc
	mem_pc
	mem_en
	mem_br_flag
MEM/WB 流水线寄	mem_ctrl_op
存器	mem_dst_addr
	mem_gpr_we_
	mem_exp_code
	mem_out
	if_busy

流水线的 状态	ld_hazard
	mem_busy
	if_stall
延迟信号	id_stall
延 及信号	ex_stall
	mem_stall
	if_flush
	id_flush
刷新信号	ex_flush
	mem_flush
	new_pc

	int_en
	pre_exe_mode
	pre_int_en
控制寄存	ерс
器	exp_vector
	exp_code
	dly_flag

	mask
内部信号	pre_pc
	br_flag

CPU 控制单元针对各个流水线阶段的延迟和刷新操作进行控制。代码 1-33 展示了生成 CPU 控制信号的部分代码。

代码 **1-33 CPU** 控制信号的生成 (**ctrl.v**)



[I] 延迟信号的赋值

延迟信号(stall)在 IF 阶段的忙信号 (if_busy)或 MEM 阶段的忙信号 (mem_busy)任何

一个有效时有效。由于 IF 阶段发生 Load 冒险时也需要延迟,最终延迟信号(stall)与 Load 冒险信号(ld_hazard)进行 OR 运算。

[Ⅱ]刷新信号的赋值

由于 ID 阶段发生 Load 冒险时也需要 刷新流水线,最终刷 新信号(flush)与 Load 冒险信号 (ld_hazard)进行 OR 运算。

[III]流水线刷新的控制

此处生成刷新信号 (flush)与刷新时的 新的 PC 寄存器 (new_pc)值。 (1)处指定默认 值。初始化刷新信号 (flush)为无效、新 PC 寄存器 (new_pc)值为 0。

在有异常发生时,刷新流水线并将 CPU的执行引入异常处理程序。(2)处使能刷新信号(flush)并将异常向量

(exp_vector) 写入

新 PC 寄存器 (new_pc)。

通过执行 EXRT 指令 从异常恢复时,刷新 流水线,并从异常程 序计数器重启程序。 (3)处使能刷新信 号(flush)并将异常 程序计数器(epc) 写入新 PC 寄存器 (new_pc)。

执行 WRCR 指令对 控制寄存器进行写入 操作后,之后的指令 需要反映出 CPU 状 态变化。因此要将流 水线刷新一次再执行 下面的指令。由于 MEM 阶段的 PC (mem_pc) 指向 WRCR 指令的下一个 地址, 因此从 MEM 阶段的 PC (mem_pc) 的地 址开始执行。(4) 处使能刷新信号 (flush) 并将 MEM 阶段的 PC (mem_pc) 写入 新 PC 寄存器 (new_pc) 。

接下来,我们通过代码 1-34 对中断检测部分程序进行说明。

代码 **1-34** 中断检测 (ctrl.v)

[I] 中断检测

中断有效信号 (int_en)有效,并 且有任何中断请求发 生的情况下,中断检 测信号(int_detect) 有效。中断请求信号 (irq)会在中断屏蔽 (mask)对应位为1 时被屏蔽。

应用屏蔽时,将有的 屏蔽(mask)所有位 翻转,并与中断请 信号(irq)进行 AND运算。然后取 信号 OR 运算。 所有位的 OR 运算第一 果为 1 的中断请求中 果为 1 的人测出有 是,则检测出有 一

图 1-113 展示了中断 检测的逻辑。中断屏 蔽(mask)可以针对 各个中断请求(irq) 设置有效或无效。而 中断有效信号则可以 设置全体中断是否有 效。

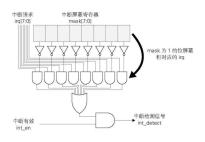
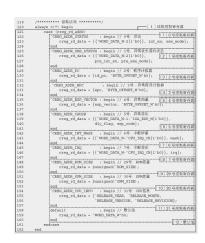


图 **1-113** 中断检测 的逻辑

接下来,通过代码 1-35 对读取控制寄存器部分的源程序进行说明。

代码 **1-35** 控制寄存器的读取(**ctrl.v**)



[Ⅰ]读取控制寄存 器

控制寄存器的读取基 本上只是根据输入的 读取地址

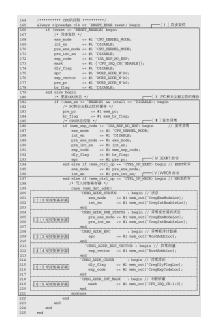
(creg_rd_addr)将 对应控制寄存器的值 输出到数据读取信号 (creg_rd_data)。

各控制寄存器位宽不同,未使用的位用 0 填充,然后输出到数据读取信号

(creg_rd_data)。 (1)处~(11)处分 别对控制寄存器 0~7、29~31进行读 取。(12)处将0作 为默认值代入。

对 CPU 进行控制部分的源程序如代码 1-36 所示。这部分进行控制寄存器的写入、异常的发生和恢复等控制操作。

代码 **1-36 CPU** 的控制(**ctrl.v**)



[I] 异步复位

复位信号 (reset) 有

效时,全部寄存器将被初始化。复位时, 执行模式 (exe_mode)被设为 内核模式 (CPU_KERNEL_MC 中断有效信号 (int_en)被设为无 中断解被设置为无 中断解被设置为全 解本。其他效、 等存器被设为无数据初始化为 0。

[II] PC 和分支标志位的保存

在 MEM 阶段的流水 线寄存器的数据有 效,且没有延迟发生的情况下,在此处之 前 PC(pre_pc)保存 MEM 阶段的 PC(mem_pc)、分 支标志位(br_flag)。 失存 MEM 阶段的 支标志位 (mem_br_flag)。 这些信息在异常发生时使用。

[III] 发生异常

MEM 阶段异常代码 (mem_exp_code) 被设置时说明有异常 发生。异常发生时,

将当前执行模式 (exe_mode) 保存到 之前执行模式 (pre exe mode) 中,并将当前中断有 效信号(int en)保 存到之前中断有效信 号 (pre_int_en) 中。然后,将执行模 式 (exe mode) 设置 为内核模式 (CPU KERNEL MC 中断有效信号 (int en) 设置为无 效。将 MEM 阶段异 常代码 (mem_exp_code) 保存到异常代码

保存到异常代码 (exp_code)中。分 支标志位(br_flag) 有效时,因为前一条 指令为分支指令,需 要再此插入延迟间隙 标志位

(dly_flag)。最后,由于流水线寄存器中保存的 PC 指向下一条指令的地址,需要将之前的PC(pre_pc)值代入EPC(epc)。

「IV] EXRT 指令

从异常恢复时执行 EXRT 指令。从异常 恢复时,将异常发生

时备份的之前执行模式(pre_exe_mode)恢复到当前执行模式(医是_mode),之前中断有效信号(pre_int_en)恢复到当前中断有效信号(int_en),将 CPU 恢复到异常发生前的状态。

[V] WRCR 指令

执行 WRCR 指令可将 MEM 阶段输出信号(mem_out)存入写入地址(mem_dst_addr)指定的控制寄存器中。(1)~(6)处对相应的控制寄存器执行写入操作。

• CPU 顶层模块

最后将流水线的各个阶段模块及其通用寄存器、CPU 控制模块以及 SPM 相连接,以及 SPM 相连接,就完成了整个 CPU 的项层模块由名为 cpu 的顶层模块构成。CPU 的顶层模块构成。CPU 的顶层模块构成。CPU 的顶层模块构成。T-114 所示。



图 **1-114 CPU** 顶层 模块连接图

1.8.4 小结

专栏

计算机架构相关 书籍

> > (中文译

名:《边做 边学计算机 架构》)

本书使用 Verilog HDL 制作 16 位的 CPU,对计 算机架构进 行了系统、 通俗易懂的 讲解。本书 难度属于大 学和大专教 材水平,适 合初学者学 习。阅读本 书虽然需要 一定的编程 和电路基 础,但作者 对基础知识 的讲解简单 易懂,可以 作为读者们 的第一本计 算机架构入 门书。

コの構第4 設計 デ?A? 以ッタ、ジューとスの構第イデ?A?スのは、スの

ン?L?へネ シー著、成 田光彰翻 訳、日経 **BP** 社)

(《成硬接第版继極等工社中计与件口4)昌、译业,版机计软原 , 樊建机版:组:件书 康晓峰械

2012.1)

本书是系统 讲解计算机 架构的世界 名著。作者 David A. Patterson 与 John L. Hennessy 是 计算机架构 的世界级权 威,他们写 的教科书被 作为计算机 基础教育的 标准用书。 本书是计算 机架构专业

人员必备的 一本著作。

•

•

• **1.9 I/O** 的设 计与实现

本节讲解 I/O 的设计 与实现。本节要设计 的 I/O 有三种,分别 是测量时间用的定时 器、串口通信规范 **UART** (Universal **Asynchronous** Receiver Transmitter),以 及控制 LED、开关 用的 **GPIO** (General **Purpose Input** Output)。它们都 是最基本的 I/O,几 乎所有计算机都配有 全部三种或其中一部 分 I/O 接口。

1.9.1 定时器

• 什么是定时器

定时器是用来测 量时间的装置。 和我们日常使用 的厨房定时器、 起床闹钟的功能 完全相同。计算 机可以利用定时 器实现时间测 量、周期性处 理、超时判断等 许多用途。定时 器通过软件设置 定时的时长并启 动,经过设定时 间后引发 CPU 中断请求。

• 定时器的设计

我们将要设计的 定时器具有两种 动作模式:一种 是经过设定时间 后向 CPU 请求 一次中断即完成 操作的单次定时 模式,另一种是 每经过设定时间 就向 CPU 请求 一次中断的循环 定时模式。单次 定时器在只进行 一次时间测量时 使用,循环定时 器在需要执行周 期性操作时使 用。

表 **1-57** 定时器 控制寄存器

控制寄存器0:控制寄存器

[0]: 起始 位(S)

该位用来控制定时器的开/关。该位为1时器开始的一个。该位为1时器开始计数。

[1]: 模式 位 (**M**)

该位用来设

置定时器的 动作模式。 该位为1时 定时器为循 环定时模式。

控制寄存器1:中断寄存器

[0]: 中断 位(I)

• 控制寄存器 2: 最大值 寄存器

> [**31:0**] : 最 大值 (**EXPR_V**)

示定时时间 到。

• 控制寄存器 3: 计数器 寄存器

> [31:0]: 计 数器 **(COUNTE**

该寄存器为 定时器。计时 数器。计时 存器的值 始增长。

• 定时器的实现

我们设计的定时器由一个被称为timer的模块构成。定时器的框图如图 1-115 所示。该模块由记录经过时间计数器

(COUNTER)、 表示定时时间的 计数最大值寄存 器

(EXPR_VAL) 控制定时器启动 和动作模式的起 始位(S)与模 式位(M)以及 通知计时完成的

中断位(I)构成。

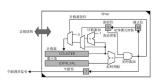


图 **1-115** 定时器框图

起始位设为1时 计数开始, 计数 到达最大值后计 数器被初始化, 并设置中断位为 1。此时如果模 式位为单次定时 模式,则会将起 始位清零。如果 模式位为循环定 时模式,则自动 进入下一个计数 周期。定时器的 宏一览如表 1-58 所示,信号线一 览如表 1-59 所 示,源程序如代 码 1-37 所示。

表 1-58 宏一览 (timer.h)

宏名称

TIMER_ADDR_W

TimerAddrBus
TimerAddrLoc
TIMER_ADDR_CTF
TIMER_ADDR_INT
TIMER_ADDR_EXF
TIMER_ADDR_COU

	TimerStartLoc			
	Tim	nerModeLoc		
	TIM	∕IER_MOD	E_(ONI
	TIM	∕IER_MOD	E_F	PER
	Tim	nerIrqLoc		
表 1-59 信号线 一览(timer.v)				
	分组	信号名	信号类型	要 抹 学 弄
	时钟	clk	输入端口	wi

	1	
reset	输入端口	wi
cs_	输入端口	wi
as_	输入端口	wi
rw	输入端口	wi
addr	输入端口	wi
wr_data	输入端口	wi
rd_data	输出端口	reş
rdy_	输出端口	reş
irq	输出端口	reş
	cs_ as_ rw addr wr_data rd_data	reset 端口 输入端口 输入端口 输入端口 输入端口 输入端口 输入端口 输入端口 输

控制寄存器 0	mode	内部信号	reş
	start	内部信号	reş
控制寄存器 2	expr_val	内部信号	reş
控制寄存器3	counter	内部信号	reş
内部信号	expr_flag	内部信号	wi

代码 **1-37** 定时 器控制逻辑 (**timer.v**)



[I] 计时完成 标志位的生成

起始位(start) 为有效,且计数 器(counter)值 等于计数最大值 (expr_val) 时,计时完成标 志位 (expr_flag)为 高电平。

[II] 异步复位

复位信号 (reset)有效时 对寄存器进行初 始化。初始化时 全部控制信号设

为无效,数据信号设为0。

[III] 就绪信号 的生成

[IV] 读取访问

片选信号 (cs)与地址 选通信号 (as) 同时有 效、且读/写信 号(rw)为读取 (READ) 时, 发生读取访问操 作。(1)处对 地址信号 (rd data) 进行 解码,并将对 应控制寄存器的 值输出到数据读 取信号 (rd_data) 。

(2) 处针对无 访问情况下,向

数据读取信号输 出 0。

[V] 写入访问

- (4)处对起始位(start)进行控制。定时器动作模式(mode)为单次定时模式(TIMER_MODI时,计时完成后将起始位(start)清零。
- (5)处对中断请求(irq)进行控制。中断请求(irq)比写入控制寄存器 1 操作优先级高。这

样设计是为了防止在写入同时计时结束的情况发生时,无法获取来自定时器的中断请求。

(9) 处在计时 完成时将计数器 (counter) 初始 化为 0。(10) 处对计数器 (counter) 数器 定时器启动,并数最后动,大型 定时器启动,大量 时进行累加操 作。

1.9.2 **UART**

• 什么是 UART

UART同发信业体信串标一地式无可发信U种发线通。接输为是步送装不使两进信数一的串起接串

通信。图 1-116 所示的 是 UART 通信的波形 图。

\$4.00 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550 | 550

图 **1-116 UART** 通信 波形图

起止式同步 通信是指在 发送的数据 前添加表示 通信开始的 起始位

(L)、在 数据末尾添 加表示通信 结束的停止 位(H)的 通信方式。 空闲时总是 输出停止 位。数据从 LSB 一端开 始按顺序输 出,并可以 选择添加奇 偶校验位, 最后输出停 止位。数据 传输单位为 7位或8 位。

UART 的通 信速率用波 特率(baud rate)来表 示。波特率 指的是信号 被调制以后 的变化率, 即单位时间 内载波变化 的次数。用 于波特率计 算的信号除 了数据位, 也包含起始 位、停止 位、奇偶校 验位, 因此 波特率与单 纯的数据传 输速率是不 同的。 UART 常用 的波特率有 9 600 baud 19 200 bauds 38 400 baud

专栏

等。

UART 实例

计算机 背面面

板通常 有一个 与 VGA 端口相 似的9 针接口 (DE-9 接 头), 如图 1-117 所 示。这 个端口 被称为 RS-232 (Reco Standard 232) 或串 口,是 一种 **UART** 标准的 实现。 RS-232 是可以 用来连 接调制 解调器 等计算 机周边 设备, 或者作 为控制 台接口 **1** 。

图 1-117 RS-232(DE-9 接 头)

• **UART** 的设计

以将UART波 400 baud、比偶停比T存为 1-60 baud、比偶停比T存 40 数特校止特的器所 1-60 据、位。控如

表 **1-60 UART** 控制 寄存器

控制寄存器 0: 状态寄存器

[0]:接收

完成中断位 (RI)

[1]: 发送 完成中断位 (TI)

[2]:接收 中标志位 (**RB**)

该位在数据 接收时为高 电平。

[3]: 发送 中标志位 (TB)

该位在数据 发送时为高 电平。

控制寄 存器 1 数据 存器

[**7:0**]: 收 发的数据 (**DATA**)

• **UART** 的实现

我UART 1-61 模UART 所构T图和T块块块行的构T图所T块块块行的构计由示成的1-。发接控块体质的表的。框 。发接控以体层。

UART 使用 的宏一览如 表 1-62 所 示。

表 **1-61 UART** 的模 块

模块名	文化
uart	uart.v
uart_tx	uart_
uart_rx	uart_
uart_ctrl	uart_

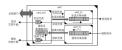


图 **1-118 UART** 的框 图

表 1-62 宏 一览 (uart.h)

宏名

UART_DIV_R

UART_DIV_C

UartDivCntBus

UartAddrBus

UART_ADDR\

UartAddrLoc

UART_ADDR\

UART_ADDR\

UartCtrlIrqRx

UartCtrlIrqTx

UartCtrlBusyRx

UartCtrlBusyTx

UartStateBus

 $UART \backslash _STATE$

UART_STATE

UART_STATE

UartBitCntBus

UART_BIT_C

UART_BIT_C

UART_BIT_C

UART_BIT_C

UART_START

UART_STOP_

发送模块

发块为有态发块态图送设一限机送的迁如模计个状。模状移图

1-119 所示。 该模块 的状态 有"空 闲状 态"和" 送状 态"两 种。处 于发送 状态 时,依 据比特 计数器 对下一 个发送 的数据 进行控 制。



图 **1-119** 发送模 块的状 态迁移

发块闲时果开 发短点

号到 来,则 保存发 送的数 据并开 始发 送。发 送模块 基于输 入的时 钟生成 需要的 波特 率。相 对于 **UART** 常用的 波特 率,例 如 38 400 baud, 向电路 输入的 时钟高 达数十 MHz, 因此需 要对输 入的时 钟进行 分频来 生成波 特率。 时钟的 分频比 率计算

公为入频波率分用器到的率式输钟: 使数得要

发送状 态时, 每当分 频计数 器计满 预定次 数时、 发送下 一个比 特数 据。发 送状态 中,依 次发送 起始 位、数 据 0~7 位、停 止位, 最后将 比特计 数器清 零并返 回空闲

状发毕出完号送时输信态送后发成。信模出号。完输送信发号块忙。

发块号表所源如1-38模信如3,序码所

表 1-63 信 号线一 览 (uart_

分组

控制信号	t
	t
	t
	t
UART 发送 信号	t
内部信号	st
	d
	b
	sl

代码 1-38 UART 发送模 块 (uart_



[I] 发送志 号的生 成

[II] 异步复 位

复号(reset) 有时行器化作。 等初操。

[III]

空闲状态

(1) 处在空 闲状态 时,如 果开始 信号 (tx_sta 到来, 则将发 送的数 据 (tx_da 保存到 移位寄 存器 (sh_re 输出开 始位、 状态迁 移到发 送状态 (UAR

[IV] 发送状 态

(3) 和(8) 处特行抱。 处特行的。

频用计 数器 (div_c 使用倒 数方式 (8), 当值倒 数到0 时发送 下一比 特数据 (3)。 (4) 处对发 送的比 特数据 进行控 制。因 为在迁 移到发 送状态 (UAR' 时开始 发送起 始位, 所以此 处发送 剩下的 数据位 和停止 位。

(5) 处在发 送完数 据的 MSB(1)

七位) 后,发 送停止 位并递 增比特 计数器 (bit_cr (6) 处当停 止位发 送完 毕,比 特计数 器 (bit_cr 归零初 始化, 使能发 送完成 信号 (tx_en 并将状 态迁移 到空闲 状态 (UAR' 最后剩 下的一 种情况 为数据 的发 送, (7) 处发送 数据比 特,并 递增比

特计数 器 (bit_cr 因为数 据从 LSB 开 始发 送, 所 以在发 送的同 时移位 寄存器 (sh_re 向右移 动一 位。

接收模块

图 1-120 UART 接收示 例

[**1**] 起始位 的检测

接号L为到位。

[**2**] 起始位 的中心

在出位始波的期确始中检起时测特半,定位心测始开量率周以起的。

[**3**] 数据接 收开始

从起始 位的中

心算率个位然 LSB 接据计特 1 期,从开收。

[**4**] 数据接 收完成

数收MSB MSB,示接成。

[**5**] 停止位 的接收

检测接号 使波 用比率

高的频 率进行 采样, 由于检 测到起 始位的 同时开 始同步 (起止 式同 步)传 输,因 此没有 专门的 同步信 号也可 以传输 数据。 为了准 确接收 数据, 采样时 钟必须 具有比 波特率 充分高 的频 率。一 般使用 比波特 率高 16倍 的时钟 进行采 样。采 用这个 频率是

因早的UM片了倍样钟为开 RT芯用 系

接收模 块和发 送模块 一样, 也是基 于有限 状态机 制作 的。接 收模块 的状态 迁移图 如图 1-121 所 示。该 模块有 空闲和 接收两 个状 态,接 收状态 下依据 比特计 数器控 制数据 的接

收。



图 **1-121** 接收模 块的状 态 图

接收模 块在空 闲状态 下检测 到起始 位后, 开始接 收信 号。此 时,分 频计数 器中记 入波特 率的半 周期。 第一次 分频计 数器计 数满时 的位置 为起始 位的中 心。此 后每过 一个周

期接收 一个数 据位。

接收状 态下, 依次接 收起始 位、数 据 1~8 位、停 止位。 当正确 接收到 停止位 (H)后使能 接收完 成信 号、将 接收到 的数据 输出、 为下次 接收数 据将分 频计数 器设置 为半周 期,并 返回空 闲状 态。

接收到 的停止 位为错

误的值 (Γ) 时称为 帧错误 (Fram Error) 此时将 接收到 的数据 废弃并 返回空 闲状 态。帧 错误是 指帧的 同步不 成功的 状态。 接收信 号时模 块输出 忙信 号。

接块号览1-64,字码所。模信一表所源如1-

表 1-64 信

号线一 览 (uart_

分组	1
时钟复位	c
	rŧ
控制信号	rz
	rz
	rz
UART 接收 信号	rz
内部 信号	S1
	d
	b

代码 1-39 UART 接收模 块 (uart_



[I] 接收志信 据志的生 成

[II] 异步复 位

复位信 号 (reset)

有效 时行器初 器初操 作。

[III] 空闲状 态

(处闲下到位转接态()在状检起时移收 UAR

(处据完使收信(后接成进除)在接毕能完号 rx,收信行。数收、接成 en对完号清

[IV] 接收状

(3) 和 (8) 处对波 特率进 行控 制。分 频用计 数器 (div_c 使用倒 数方式 **(8)**, 当值倒 数到 0 时接收 下一比 特数据 (3) 。 (4) 处对接 收的比 特数据

在(处到位进下完的)坡止,以收后

进行控

制。

(6) 处在接 收完成 时对帧 错误进 行检 测。当 停止位 为H 之外的 错误值 时,视 为发生 帧错 误。当 停止位 为H 时,生 成接收 完成信 号

(7) 处接收 数据比 特并递 增比特 计数器 (bit_cr 移位寄 存器 (sh_re 向右移 位1比 特、将 接收到 的数据 插入 $MSB_{\,\circ}$ 因为从 LSB 端 开始依 次接收 数据, 最初接 收的数 据经过 不断移 位,最 终将移 位到 LSB 的 位置。

控制模块

UART 的控制 模块用 来控制 UART 的信号 收发和 控制寄 存器的 访问。 **UART** 控制模 块的信 号线一 览如表 1-65 所 示,源 程序如 代码 1-40 所 示。

表 **1**-65 信 号线一 览 (uart_

分组	信号
时钟复位	clk
	reset

	Ī
总线接口	cs_
	as_
	rw
	addr
	wr_c
	rd_d
	rdy_
中断	irq_r
	irq_t

控制信号	rx_b
	rx_eı
	rx_da
	tx_bı
	tx_eı
	tx_st
	tx_da
控制寄存器1	rx_b

代码 1-**40**

UART 控制模 块 (uart_



[I] 异步复 位

复号(reset) 有时行器化作。 一种,等初操。

[Ⅱ] 就绪信 号的生 成

当信(和选号(同片号 cs址通 as时

来始访使绪(其况绪(无时总问能信 rd处下信 rdy),就号(情就号))。

[III] 读取访 问

当片选 信号 (cs_) 和地址 选通信 号 (as_) 同时到 来,且 读/写 信号 (r_{W}) 为读取 (REAl 时开始 读取访 问。 (1)

处的 case 语 句对地

址进行 解码, 地址 (addr) 对应的 控制寄 存器的 值会作 为读出 的数据 (rd_da 被输 出。 (2) 处读取 控制寄 存器 0, (3) 处读取 控制寄 存器 1。控 制寄存 器1的 收发数 据从接 收缓冲 X (rx_bu 读取。 (4) 处在无 访问时 向读取 的数据

信号

(rd_da 输出 0。

[IV] 写入访 问

当信(cs_) 治号(加选号(加进信 as_)

同来, 月

(rw) 为写入 (WRI]

时,对 地址信 号

(addr) 指向的 控制寄 存器 入数 据。

(6)

和

(8) 处对寄 存器 0 进行写

入操 作。

(5) 处对发 送完成 中断进 行控 制。发 送完成 时的处 理比写 入控制 寄存器 0的操 作优先 级高。 (7) 处对接 收完成 中断进 行控 制。接 收完成 时的处 理比写 入控制 寄存器 0的操 作优先 级高。 (5) 和 (7) 处的中 断比来

自总线

的访先理为防入时完情丢断写问处,了止的发成况失。入优善是在写同送的下中

(9) 处对控 制寄存 器1进 行写入 操作。 当有数 据写入 控制寄 存器1 时,写 入的数 据作为 发送的 数据 (tx_da 输出, 同时输 出发送 开始信 号 (tx_sta (10)

无写入

时在 (11) 处清除 发送的 数据 (tx_da 与发送 开始信 号 (tx_sta 当接收 完成信 号 (rx_en 有效 时,在 (12) 处将接 收的数 据 (rx_da 放入接 收用数 据缓冲 X (rx_bu

 顶层模 块

顶块连送块收和层用接模、模控

模块。 顶层模 块的端 口连接 图如图 1-122 所示。 控制模 块与总 线接 口、中 断请求 信号相 连接。 控制模 块与发 送模 块、控 制模块 与接收 模块间 通过控 制信号 相连 接。发 送模块 与 **UART** 的发送 信号相 连、接 收模块 与 **UART** 接收信 号相 连。

THE REPORT OF TH

图 **1-122 UART** 顶层模口 连接图

> 1 Console 公田外用计或设行的接。 | 注

1.9.3 **GPIO**

• 什么是 GF

GP In Ou是以位为单

• **GF** 的 设 计

我们设计的 GP有输入专用端 口输出专用端

		•

		•

		•

-

图

1-124

GF 框图

GP

表 1-68 信号线一览(

分组

时

码 1- 41 GF的输入输

[输入输出信号的定义 此处定义输入输出端口所用的信号输入输出端

当 前

[] 输

态用

z表示

接下来我们来说明总线访问的控制部分

当片选信号(和地址选通信号(同时

[写入访问 当片选信号(和地址

•

•

• 1. A Sc 整体连接 本节中我们将做好的CP内存以及

各种IC使用总线连接完成AZ S的制作首先我们要制作名为 chi 的模块该模块中 使用总线

连接 本节我们将做好的CRC定时器UG以及

234567

IRO

IRO

IRO

IRO

IRO

IRO

IRO

1.1时钟模块的实现

时钟模

chi

[复位信号的生成

亏的生成 (处生成D的复位信号(因为DQ

[**D**的实例化 此处实例化D模块并进行信号连接

•

•

• 1. A Sc 的仿真 本

仍真 本节对制作完成的AS的仿真进行

		•

		•

Tes Té使用的宏一览如表 1-76所示信号线一览如表 1-77所示 表

76 宏一览(

时纪复位

的监测 当输入端口值变化时此

[输出端口的监测

口值变化时此

U接收模块实例化作为U模型使用由

Entra et l'Arrente Entra et l'Ar [测试用例

() 处对信号线进行初

• **1.** 本章总结 本章中我们设计了CP内存IC以及它们的

穿2章电路秘的设计与带

•

2.2 电路板规格

2.3 元件选型

2.4 电路设计

2.5 布局设计

2.6 制

2.使用电路板制造服务

•

•

• **2.**:

在第2章中我们将设计和制作电路板来运行第1

照片2-1本书中制作的电路板样板 制作流

•

• 2. 电路板规格

凡格 本节将确定电路板的规格制定AEV

AZ Ev 的

A Z v 由一块 FP 电路板和一块电源电路

除了FP电路板还需要设计

驱动 Spi 需要的电源有 1.2.3.三种其中

1.2

是用作 FP

2.2 外围电路的选定 板上的外围电路与FP的用户IC相连接用户IC是指用户

AZV上搭载的外围电路要可以充分发

Ī

: : :

]

; ;

• **2.** 元件选型

口选 型 本节将对AEN所使用的元件进行选型并

		•

	•

		•

		•

		•

		•



•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •

•
•
•
• •